An Annual Progress Report
Contract No. N00014-86-K-0245
October 1, 1989 - September 30, 1990

THE STARLITE PROJECT

Applied Math and Computer Science
Dr. James G. Smith
Program Manager, Code 1211

Computer Science Division
Dr. Gary Koob
Program Manager, Code 1133

Submitted to:

Director
Naval Research Laboratory
Washington, DC 20375

Attention: Code 2627

Submitted by:

R. P. Cook
Associate Professor

S. H. Son
Assistant Professor

Report No. UVA/525410/CS91/104
September 1990

SCHOOL OF
ENGINEERING & APPLIED SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

University of Virginia
Thornton Hall
Charlottesville, VA 22903

UNIVERSITY OF VIRGINIA
School of Engineering and Applied Science

The University of Virginia's School of Engineering and Applied Science has an undergraduate enrollment of approximately 1,500 students with a graduate enrollment of approximately 600. There are 160 faculty members, a majority of whom conduct research in addition to teaching.

Research is a vital part of the educational program and interests parallel academic specialties. These range from the classical engineering disciplines of Chemical, Civil, Electrical, and Mechanical and Aerospace to newer, more specialized fields of Applied Mechanics, Biomedical Engineering, Systems Engineering, Materials Science, Nuclear Engineering and Engineering Physics, Applied Mathematics and Computer Science. Within these disciplines there are well equipped laboratories for conducting highly specialized research. All departments offer the doctorate; Biomedical and Materials Science grant only graduate degrees. In addition, courses in the humanities are offered within the School.

The University of Virginia (which includes approximately 2,000 faculty and a total of full-time student enrollment of about 17,000), also offers professional degrees under the schools of Architecture, Law, Medicine, Nursing, Commerce, Business Administration, and Education. In addition, the College of Arts and Sciences houses departments of Mathematics, Physics, Chemistry and others relevant to the engineering research program. The School of Engineering and Applied Science is an integral part of this University community which provides opportunities for interdisciplinary work in pursuit of the basic goals of education, research, and public service.

An Annual Progress Report
Contract No. N00014-86-K-0245
October 1, 1989 - September 30, 1990

THE STARLITE PROJECT

Applied Math and Computer Science
Dr. James G. Smith
Program Manager, Code 1211

Computer Science Division
Dr. Gary Koob
Program Manager, Code 1133

Submitted to:

Director
Naval Research Laboratory
Washington, DC 20375

Attention: Code 2627

Submitted by:

R. P. Cook
Associate Professor

S. H. Son
Assistant Professor

A-1

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/525410/CS91/104
September 1990

Copy No. 15

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Annual: Oct. 1, 1989 - Sept. 30, 1990

4. TITLE AND SUBTITLE

The StarLite Project

5. FUNDING NUMBERS

N00014-86-K-0245 P00002

6. AUTHOR(S)

R. P. Cook, S. H. Son

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22901

8. PERFORMING ORGANIZATION REPORT NUMBER

UVA/525410/CS91/104

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research Resident Representative
818 Connecticut Avenue, N. W. Eighth Floor
Washington, DC 20006

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The StarLite Project has the goal of constructing a program library for real-time applications. The initial focus of the project is on operating system and database support. The project also involves the construction of a prototyping environment that supports experimentation with concurrent and distributed algorithms in a host environment before down-loading to a target system for performance testing.

The components of the project include a Modula-2 compiler, a symbolic Modula-2 debugger, an interpreter/runtime package, the Pheonix operating system, the meta-file system, a visual simulation package, a database system, and documentation.

14. SUBJECT TERMS	15. NUMBER OF PAGES
StarLite Project, Modula-2 compiler, Modula-2 debugger, Pheonix operating system	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (890104 Draft)
Prescribed by ANSI Std 239-18
298-01

TABLE OF CONTENTS

Robert P. Cook and Sang H. Son
University of Virginia
(804) 982-2215
cook or son@cs.virginia.edu
The StarLite Project
N00014-86-K-0245
10/1/89 - 9/30/90

1. Productivity Measures

- Refereed papers submitted but not yet published: 8
- Refereed papers published: 11
- Unrefereed reports and articles: 5
- Books or parts thereof submitted but not yet published: 2
- Books or parts thereof published: 1
- Patents filed but not yet granted: 0
- Patents granted: 0
- Invited presentations: 4
- Contributed presentations: 9
- Honors received: 12
- Prizes or awards received: 0
- Promotions obtained: 0
- Graduate students supported: 12
- Post-Docs supported: 0
- Minorities supported: 2

Robert P. Cook and Sang H. Son
University of Virginia
(804) 982-2215
cook or son@cs.virginia.edu
The StarLite Project
N00014-86-K-0245
10/1/89 - 9/30/90

2. Summary of Technical Progress

This section contains a summary of technical progress on the StarLite Project.

2.1. The StarLite Environment

The StarLite integrated programming environment is designed to support research in real-time database, operating system, and network technology. The StarLite library synthesizes intellectual effort in the form of software components that can be transferred, examined, learned, or experimented with by the community at large. Thus, an abstraction, such as sorting, may be associated with interfaces, implementations, performance ratings, text, hypertext, video animations, sample programs, test cases, formal specifications, modification history, etc.

StarLite has been used to implement a real-time UNIX (Phoenix), a distributed database system, a parallel programming interface identical to WorkCrews used on the DEC SRC FireFly, and network protocols. It has also been used to support graduate and undergraduate courses in operating systems and database technology. The StarLite components include a Modula-2 compiler, an interpreter-based runtime, a SunView/X graphics package, a viewer, a simulation package, movie system, profiler, browser, Prolog interpreter, relational database system, and a software reuse library (250 modules, 100,000 lines). New tools under development include a modelling system and an algorithm animation package.

Over the past year, we have converted all the tools to run under X as well as SunView. In fact, the conversion was designed so that all future graphics programs could be run under either windowing system without modification. Significant effort was invested in addressing the performance problems associated with running the system on an interpreter. After a number of experiments, we designed code filters to translate MCODE object modules to equivalent native code modules. The native code modules can be intermixed with MCODE modules arbitrarily and the generated code is defined in such a way that tools, such as the debugger, can be used opaquely. Performance experiments on the M680X0 filter indicate a speed-up of 10-20 and, on the SPARC filter from 15-40, over MCODE.

2.2. Database Systems

The research effort during October 1989 to September 1990 was concentrated in three areas: investigating new techniques for real-time database systems, integrating a relational database system with the real-time operating system kernel ARTS, and

developing a message-based database prototyping environment for empirical study.

2.2.1. Scheduling and Concurrency Control

Since scheduling and concurrency control is critical for satisfying timing constraints of real-time transactions, we have investigated new algorithms for transaction scheduling in real-time database systems. Real-time task scheduling methods can be extended for real-time transaction scheduling while concurrency control protocols are still needed for operation scheduling to maintain data consistency. However, the integration of the two mechanisms in real-time database systems is not straightforward. The general approach is to utilize existing concurrency control protocols, especially 2PL, and to apply time-critical transaction scheduling methods that favor more urgent transactions. Such approaches have the inherent disadvantage of being limited by the concurrency control method upon which they are based, since all existing concurrency control methods synchronize concurrent data access of transactions by the combination of two measures: blocking and roll-backs of transactions. Both are barriers to time-critical scheduling. In real-time database systems, blocking may cause priority inversion when a high priority transaction is blocked by lower priority transactions. The alternative is to abort the low priority transactions if they block a high priority transaction. This wastes the work done by the aborted transactions and in turn also has a negative effect on time-critical scheduling. One of the fundamental problems of real-time database systems is to develop real-time transaction scheduling protocols that maximize both concurrency and resource utilization subject to three constraints at the same time: data consistency, transaction correctness, and transaction deadlines.

The priority ceiling protocol, which is basically a task scheduling protocol for real-time operating systems, has been extended to the real-time database system. It is based on 2PL and employs only blocking, not roll-back, to solve conflicts. This makes it a conservative approach. We have investigated methods to apply the priority ceiling protocol as a basis for real-time locking protocol in a distributed environment. One approach to implement the priority ceiling protocol in a distributed environment is to use a global ceiling manager at a specific site. The advantage of this approach is that the temporal consistency of the database is guaranteed, since every data object maintains most up-to-date value. While this approach ensures consistency, holding locks across the network is not very attractive. Due to communication delay, locking across the network will only enforce the processing of a transaction using local data objects to be delayed until the access requests to the remote data objects are granted. This delay for synchronization, combined with the low degree of concurrency due to the strong restrictions of the priority ceiling protocol, is counter-productive in real-time database systems.

An alternative to the global ceiling manager approach is to have replicated copies of data objects. An up-to-date local copy is used as the primary copy, and remote copies are used as the secondary read-only copies. In this approach, we assume a single writer and multiple readers model for distributed data objects. This is a simple model that effectively models applications such as distributed tracking in which each radar station maintains its view and makes it available to other sites in the network.

We have investigated the performance characteristics of the global ceiling approach and the local ceiling approach with replication in a distributed environment. The real-

time database system we have prototyped for the experiment consists of three sites with fully interconnected communication network. Our performance results have illustrated the superiority of the local ceiling approach over the global ceiling approach, at least under one representative distributed real-time database and transaction model. From the results of this experimentation, we found that, even with the potential problem of temporal inconsistency (i.e., reading out of date values), the local ceiling approach is a very powerful technique for real-time concurrency control in distributed database systems. Some of our findings have been presented at the Real-Time Systems session of the International Conference on Distributed Computing Systems (June 1990).

In addition to investigating the priority ceiling protocols, we have been developing more practical real-time concurrency control protocols for distributed database systems. Our approach is based on the idea of adjusting the serialization order of active transactions dynamically, by relaxing the relationship between the serialization order and the past execution history.

One of the protocols we have been developing is a priority-dependent locking protocol, which has a flavor of both locking and optimistic approach. Our goal is to provide a locking mechanism that adjusts the serialization order, making it possible for transactions with higher priorities to be executed first so that high priority transactions are never blocked by uncommitted low priority transactions while lower priority transactions may not have to be aborted even in face of conflicting operations.

The execution of each transaction is divided into three phases: the read phase, the wait phase and the write phase. During the read phase, a transaction is executed, only reading from the database and writing to its local workspace. After it completes, it waits for its chance to commit in the wait phase. If it is committed, it switches into the write phase during which it makes all its updates permanent in the database. A transaction in any of the three phases is called an active transaction. If an active transaction is in the write phase, then it is committed and writing into the database.

Each lock contains the priority of the transaction holding the lock as well as other usual information. The locking protocol is based on the *principle* that high priority transactions should complete before lower priority transactions. This principle implies that if two transactions conflict, the higher priority transaction should precede the lower priority transaction in the serialization order. If a low priority transaction does complete before a high priority transaction, it is required to wait until it is sure that its commitment will not lead to the abort of a higher priority transaction.

2.2.2. Integration of a Relational Database with ARTS

A database system must operate in the context of available operating system services. In other words, database operations need to be coherent with the operating system, because correct functioning and timing behavior of database control algorithms depends on the services of the underlying operating system. As pointed out by Stonebraker, operating system services in many systems are not appropriate for support of database functions. In many areas such as buffer management, recovery, and consistency control, operating system facilities have to be duplicated by database systems because they are too slow or inappropriate. An environment for database systems development, therefore, must provide facilities to support operating system functions and integrate them with database systems for experimentation.

The ARTS is a real-time operating system kernel being developed by the researchers at the Carnegie-Mellon University. The goal of the ARTS OS is to provide a predictable, analyzable, and reliable distributed real-time computing environment. We have been working closely with the ARTS developers and Pat Watson at the IBM Federal Systems Division to develop a relational real-time database system, called RTDB, and to integrate it with ARTS. Initially the RTDB runs on UNIX, where all relations are stored as files on disk. It was designed as a single-user program, and hence the code was not necessarily re-entrant. Many changes were made to have the RTDB running on the ARTS in a server-client mode in which the server accepts requests from multiple clients possibly on different machines. Our initial efforts have resulted in a single-threaded server that can accept requests from multiple clients. Each client is an object that submits a series of requests to access the database.

We have developed two different kinds of clients for the RTDB. One is an interactive command parser/request generator that makes requests to the server on behalf of the user. This client looks and behaves just like the single-user database manager running on Unix. It is possible to run the client without knowing that any interaction between server and client is occurring. The other client is a transaction generating client. It represents a real-time process that needs to make database access requests.

We are investigating methods to have a server with multiple threads. The ARTS kernel supports lightweight processes which means that a single object can have many active control threads at the same time. This is implemented using shared memory address space for the threads. A server that takes advantage of this feature can be designed in a number of different ways. There can be threads that accept only requests for a certain type of operation; or there can be threads that accept requests for any operation of a certain priority; or there can be threads that accept any operation of any priority -- this would be just like having more than one complete server. In addition, we are expanding the functionalities that can be provided by the real-time relational database manager.

2.2.3. Development of A Database Prototyping Tool

A prototyping tool to implement database technology should be flexible and organized in a modular fashion to provide enhanced experimentation capability. A user should be able to specify system configurations such as the number of sites, network topology, the number and locations of processes, the number and locations of resources, and the interaction among processes. We use the client/server paradigm for process interaction in our prototyping tool. The system consists of a set of clients and servers, which are processes that cooperate for the purpose of transaction processing. Each server provides a service to the clients of the system, where a client can request a service by sending a request message to the corresponding server. Our method for developing a database prototyping environment has been presented at the International Conference on Systems Integration (April 1990).

We have enhanced the previous version of the prototyping tool running on a Sun workstation. The current prototyping tool provides concurrent transaction execution facilities, including two-phase locking and timestamp ordering as underlying synchronization mechanisms. A series of experiments have been performed to test the correctness of the design and validity of the current implementation of those two

synchronization mechanisms. As a general rule, we found that transaction response time, in both mechanisms, increases with the increase of the degree of data distribution and the number of conflicts. The current prototyping tool also provides a multiversion data object control mechanism.

From a series of experiments, we found that the performance of a multiversion distributed database system is quite sensitive to the size of read-sets and write-sets of transactions. A multiversion database system outperforms the corresponding single version system when the size of the read-sets of transactions is larger than the size of the write-sets of update transactions. The ratio between read-only and update transactions also affects system performance, but this parameter is not as sensitive as the set size. Some of our findings have been presented at the International Conference on Data Engineering (February 1990).

2.3. Operating Systems

The Phoenix OS portion of the project encompasses research involving better operating system structuring techniques and better algorithms. For example, the system already includes many of the real-time enhancements proposed in the Lynx OS. Phoenix is not a kernel, such as the Spring system, but a full-function OS.

Interfaces are important because they can be standardized and because they are designed to outlive implementations and machine architectures. It is now widely accepted that the use of a procedural interface, such as the C library for UNIX, is the most advantageous method for presenting an operating system's functionality to an end user. Such an interface can be machine and language invariant. These are desirable properties given the diversity of hardware/software used by today's defense contractors.

There are two design options to choose from as the basis for an interface standard: *flat* or *layered*. An operating system with a flat interface, such as UNIX, is essentially closed; that is none of the interfaces used in the implementation can be accessed. Flat interfaces are inflexible and typically trade performance and control for generality. A layered interface specification, such as the OSI definition of ISO for computer networks, overcomes the deficiencies of the traditional, flat operating system interface designs by allowing the application engineer to choose an interface layer that most closely fits the problem to be solved. For example, if UNIX were a layered design, it would be possible for a database system to manipulate the operating system's buffer cache in a manner that has long been requested by implementors.

Access to low-level interfaces can address the performance requirements of real-time software. Another advantage of a layered design is that layers can be omitted to save space. For example, if an application does not use files, the file system could be omitted. It is also possible to implement layers in hardware to improve performance. Open interfaces also support greater control over system actions for error recovery and dynamic repair. The Phoenix operating system is based on a layered design with standard interfaces.

Two of the research issues are how to partition the layers and how to define the interfaces at each layer. To experiment with different options, we designed and implemented a UNIX-compatible operating system according to the layering principles defined by ISO. The Phoenix UNIX is proprietary in that it is not based upon nor does it

contain any code from other UNIX implementations. We have rewritten the system several times to try different layering and implementation strategies. The long-term goal of the StarLite project is to create an operating system generator that could automatically select implementations from a module library based on specified application requirements and a given target architecture. The first step toward achieving this goal is to create a library of implementation modules suitable to support applications with severe reliability requirements. The next phase of the StarLite project is creating such a library.

One of the prerequisites for experimenting with a library of operating system components is having the ability to add and delete modules or services from what we term a software backplane. Also, we felt that some composition mechanism is be necessary to achieve the goal of creating an operating system generator.

The Phoenix composition mechanism eliminates unnecessary recompilations by binding properties to processes dynamically. The method is object based but does not support inheritance. Thus, the support code is small and fast. In Phoenix, there is only one class of object, a process. Each process object can be composed of a limited number of properties that can be connected to it in any order and at any time. When the operating system boots up, each module has been statically linked to the code of the modules that it depends on. However, each module dynamically connects its data type to the process object using a low-level system call.

It is also possible to associate managers with properties. When a process object is closed, the managers are notified one at a time so that the individual fields may be closed. For example, the *exit* system call's implementation is unaware that there is a file system associated with a process. When the manager of the file-system property is invoked as the result of a process exit, it does its own cleanup by closing all open files.

Managers can also be used to monitor the actions on fields for fault detection purposes. This is somewhat equivalent to the probe points used on hardware backplanes.

Management of resources is one of the most difficult problems to solve in order to produce a full-function UNIX operating system that is capable of providing hard, reliability guarantees. Most current UNIX implementations provide none. Deadlock over internal resources can be eliminated using a number of methods. Our approach to the resource contention problem is based on priority-ordered avoidance.

This technique requires that tasks with "hard" deadlines submit claims describing future actions and timing requirements. The system then guarantees that the deadline will be met as long as the task does not exceed its computation and resource limits and neither the hardware nor software fail. Each process with "hard" deadlines must submit a claim list identifying the resources to be used and the timing requirements. The system then associates a data structure with each resource that restricts access by competing processes during critical periods. The key to success is making the avoidance test fast enough, which is achieved by using priority to totally order the necessary comparisons.

Two other areas in which we are performing experiments are a file system based on fine-grained locking and the impact of multiprocessor technology on operating systems design.

Robert P. Cook and Sang H. Son
University of Virginia
(804) 982-2215
cook or son@cs.virginia.edu
The StarLite Project
N00014-86-K-0245
10/1/89 - 9/30/90

3. Honors, Publications, Presentations

• Honors

• Cook, General Chairman, Seventh IEEE Workshop on Real-Time Software and Operating Systems, Charlottesville, VA (1990).
• Cook, Program Committee, Eighth IEEE Workshop on Real-Time Software and Operating Systems, Atlanta, GA (1991).
• Cook, Program Committee, Third Annual Workshop: Methods and Tools for Reuse, Syracuse University, (1990).
• Cook, Site Visit Team, NSF IIP Program, Purdue University.
• Cook, Member U.S. TAG to ISO SC22/WG13, Modula-2 Standard.
• Son, Program Committee Co-Chair, Workshop on Advanced Computing, (1990).
• Son, Program Committee, Sixth International Conference on Data Engineering, (1990).
• Son, Program Committee, Ninth Symposium on Reliable Distributed Systems, (1990).
• Son, Program Committee, ACM SIGMOD Conference on Management of Data (1990).
• Son, Session Chair, Ninth Symposium on Reliable Distributed Systems, Session on Distributed Databases, (1990).
• Son, Invited Paper, "Real-Time Database Systems: A New Challenge," *IEEE Data Engineering*, Vol. 13, no. 4, Special Issue on Future Directions on Database Research, December 1990 (to appear).
• Son, Invited Paper, "Prototyping Approach to Distributed Database Research," *Database Review*, Vol. 6, October 1990 (to appear).

• Refereed Publications

(1) Cook, R. P., "The StarLite Operating System," *Operating Systems for Mission-Critical Computing*, Eds. K. Gordon, P. Hwang, A. Agrawala, ACM Press, (to appear).

(2) Cook, R. P. and L. Hsu, "StarLite: A Software Education Laboratory," *Fourth SEI Conference on Software Engineering Education*, reprinted in *Software Engineering Education*, Springer-Verlag Lecture Notes in Computer Science 423, Lionel E. Deimel (Ed.), (April 1990).

(3) Cook, R. P. and H. Oh, "The StarLite Project, *Frontiers 90 Conference on Massively Parallel Computation*, (to appear).

(4) Cook, R. P., "Modula-2," *Encyclopedia of Computer Science*, (to appear).

(5) L. Sha, R. Rajkumar, S. H. Son, and C. Chang, "A Real-Time Locking Protocol," *IEEE Transactions on Computers*, (to appear).

(6) P. Shebalin, S. H. Son, and C. Chang, "An Approach to Software Safety Analysis in Distributed Systems," *Journal of Computer Systems Science and Engineering*, (to appear).

(7) S. H. Son, "Reconstruction of Distributed Databases," *Journal of Computer Systems Science and Engineering*, Vol. 5, 1990 (to appear).

(8) Son, S. H., "An Adaptive Checkpointing Scheme for Distributed Databases with Mixed Types of Transactions," *IEEE Transactions on Knowledge and Data Engineering*, December 1989, pp 450-458.

(9) Son, S. H., "An Algorithm for Non-Interfering Checkpoints and its Practicality in Distributed Database Systems," *Information Systems*, Vol. 14, No. 4, December 1989, pp 421-429.

(10) Son, S. H. and A. Agrawala, "Distributed Checkpointing for Globally Consistent States of Databases," *IEEE Transactions on Software Engineering*, Vol. 15, No. 10, October 1989, pp 1157-1167.

(11) Y. Lin and S. H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *11th Real-Time Systems Symposium*, Orlando, Florida, December 1990 (to appear).

(12) S. H. Son, "Scheduling Real-Time Transactions," *Euromicro Workshop on Real-Time Systems*, Horsholm, Denmark, June 1990, pp 25-32.

(13) S. H. Son and C. Chang, "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment," *10th International Conference on Distributed Computing Systems*, Paris, France, June 1990, pp 124-131.

(14) S. H. Son and J Lee, "Scheduling Real-Time Transactions in Distributed Database Systems," *7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, Virginia, May 1990, pp 39-43.

(15) S. H. Son and R. Cook, "StarLite: An Environment for Prototyping and Integrated Design of Distributed Real-Time Software," *Second International Conference on Computer Integrated Manufacturing*, Troy, New York, May 1990, pp 507-515.

(16) Son, S. H., "An Environment for Prototyping Real-Time Distributed Databases," *International Conference on Systems Integration*, Morristown, New Jersey, April 1990, pp 358 367.

(17) A. Grimshaw, J. Pfaltz, J. French, and S. H. Son, "Exploiting Coarse Grained Parallelism in Database Applications," *International Conference on Databases, Parallel Architectures, and Their Applications (PARBASE '90)*, Miami Beach, Florida, March 1990, pp 510-512.

(18) Son, S. H. and N. Haghighi, "Performance Evaluation of Multiversion Database Systems," *Sixth IEEE International Conference on Data Engineering*, Los Angeles, California, February 1990, pp 129-136.

(19) Son, S. H., "On Priority-Based Synchronization Protocols for Distributed Real-Time Database Systems," *IFAC/IFIP Workshop on Distributed Databases in Real-Time Control* Budapest, Hungary, October 1989, pp 67-72.

(20) Son, S. H. and R. Cook, "StarLite: An Integrated Environment for Distributed Real-Time Software," *Software Engineering Journal*, (submitted).

(21) Son, S. H. and C. Chang, "On Priority-Based Locking Protocols for Distributed Real-Time Database Systems," *IEEE Transactions on Knowledge and Data Engineering*, (submitted).

(22) S. H. Son, M. Poris, and C. Iannacone, "RTDB: An Experimental Database Manager for Real-Time Systems," *Seventh IEEE International Conference on Data Engineering*, (submitted).

- **Unrefereed Publications**

(23) Cook, R. P., "The StarLite Intellectual Reuse Project," *Third Annual Workshop: Methods & Tools for Reuse*, CASE Center, Syracuse University, (June 1990).

(24) S. H. Son and Y. Lin, "Concurrency Control using Priority-Based Locking," *Technical Report TR-90-13*, Dept. of Computer Science, University of Virginia, June 1990.

(25) S. H. Son and P. Wagle, "Scheduling using Dynamic Priority in Real-Time Database Systems," *Technical Report TR-90-03*, Dept. of Computer Science, University of Virginia, March 1990.

(26) S. H. Son and S. Chiang, "Experimental Evaluation of a Concurrent Checkpointing Algorithm," *Technical Report TR-90-01*, Dept. of Computer Science, University of Virginia, January 1990.

(27) S. H. Son and C. Chang, "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment" *Technical Report TR-89-13*, Dept. of Computer Science, University of Virginia, November 1989.

- **Presentations**

- Cook, The StarLite Project, University of Rochester.
- Son, presentation at the IFAC/IFIP Workshop on Distributed Databases in Real-Time Control (Oct. 1989).
- Son, invited talk at the Korea Information Science Society on real-time database systems (Feb. 1990).
- Son, presentation at the Euromicro Workshop on Real-Time Systems (June 1990).

- **Students**

Anthony Burrell (Ph.D. student), real-time operating system scheduling
Shi-Chin Chiang (Ph.D. student), checkpointing in distributed database systems
Lee Hsu (Ph.D student), priority-based resource management
Young-Kuk Kim (Ph.D. student), real-time database managers
Chris Koeritz(Ph.D. student), real-time file systems
Juhnyoung Lee (Ph.D. student), schedulers for real-time databases
Ying-Feng Oh (Ph.D. student), real-time multiprocessor operating systems
Jeremiah Ratner (Ph.D. student), synchronization protocols for real-time systems
Ambar Sarkar (Ph.D. student), real-time, fault-tolerant network protocols
Robert Beckinger (M.S. student), support for temporal information
David Duckworth (M.S. student), Modula-2 to C translator
Greg Fife (M.S. student), real-time, fault-tolerant broadcast protocols
Carmen Iannacone (M.S. student), multi-thread real-time database server
Spiros Kouloumbis (M.S. student), replication control
Yi Lin (M.S. student), priority-based contention protocols
Marc Poris (M.S. student), integration of a database with real-time kernel
Alan Tuten (M.S. student), relational database extension for real-time systems
Prasad Wagle (M.S. student), dynamic priority scheduling
Richard McDaniel(B.S. student), prototyping environment

Robert P. Cook and Sang H. Son
University of Virginia
(804) 982-2215
cook or son@cs.virginia.edu
The StarLite Project
N00014-86-K-0245
10/1/89 - 9/30/90

4. Transitions and DOD Interactions

- Cook, StarLite installed at the University of Rochester, San Francisco State, and the University of Pennsylvania.

- Cook and Waxman, Presentation to Raytheon Corporation, Executable Specifications, (Sept. 1990).

- Cook and Waxman, Research in the Establishment of a Simulatable Link Between an Operational Specification and a Performance Model, $49,587, funded by Hughes Aircraft Company, (June 1990).

- Cook and Weaver, Executable Specifications for Standards and Conformance Test Development, Proposal to Mr. David Hollenbeck, Space and Naval Warfare Systems Command, (July 1990).

- Cook and Son, Systems Support for an Ultra-Reliable Underwater Vehicle, Proposal to Dr. Gary Koob, Computer Science Division, Office of Naval Research, (April 1990).

- Cook and Son, presentation at the ONR Foundations of Real-Time Computing Research Initiative Workshop (Oct. 1989).

- Son, participation in the real-time database coordination meeting with Pat Watson from IBM Manassas (Nov. 1989).

- Cook, presentation to Mitre Corporation.

- Cook, presentation to SAIC.

- Cook, The StarLite Project, NOSC Code 413 DC^2 Quarterly Review (April 1990).

- Son, Real-Time Database Systems, NOSC Code 413 DC^2 Quarterly Review (April 1990).

- Cook, invited to DARPA Workshop on Software Tools for Distributed Intelligent Control Systems (July 1990).

- Son, participation at the coordination meeting with Pat Watson from IBM Manassas and Prof. Tokuda from CMU (August 1990).

Robert P. Cook and Sang H. Son
University of Virginia
(804) 982-2215
cook or son@cs.virginia.edu
The StarLite Project
N00014-86-K-0245
10/1/89 - 9/30/90

5. Software and Hardware Prototypes

The StarLite integrated programming environment is slowly evolving to the point where we could begin a national distribution. However, we still have a tremendous amount of work to do in preparing documentation for the system as it is composed of several hundred thousand lines of code. Nevertheless, we did distribute the system to several universities as beta test sites over the last few months.

APPENDIX

StarLite: An Integrated Environment for Distributed Real-Time Software

Sang H. Son and Robert P. Cook
Department of Computer Science
University of Virginia
Charlottesville. Virginia 22903

ABSTRACT

In real-time distributed systems, each component of the system must support priority-based resource management techniques in an integrated manner to satisfy the end-to-end response requirements of tasks. One of the difficulties in developing and evaluating distributed real-time software is that it takes a long time to develop a system. Also, evaluation is complicated because it involves a large number of system parameters that may change dynamically. This paper presents an integrated programming environment, called StarLite, that supports the design and investigation of distributed real-time software. Experiments in real-time locking protocols for database systems are presented as a demonstration of the effectiveness of the environment.

Index Terms - software, integrated programming environment, database, real-time

This work was supported in part by ONR under contract # N00014-88-K-0245, by IBM Federal Systems Division under University Agreement WG-249153, and by CIT contract # CIT-INF-90-011.

1. Introduction

In this paper, we report our experiences with a new integrated programming environment, StarLite. The goal of the StarLite project is to test the hypothesis that a host environment can be used to significantly accelerate the rate at which we can perform experiments in the areas of operating systems, databases, and network protocols for real-time systems. This paper discusses the scope of the StarLite project.

An *integrated programming environment* is a software and/or hardware package that supports the investigation of the properties of a software system in an environment other than that of the target hardware. Previous tools range from IBM's Virtual Machine operating system to discrete-event simulation languages and queuing analysis packages. Except for the VM approach to development, most systems support only the analysis of an abstraction of a given software system. Thus, there is the persistent problem of validating the correctness of the model.

The StarLite environment combines the benefits of the VM approach with those of modeling systems. The benefits of the VM approach are attained to the extent that development is in a host environment rather than on target hardware and that the same software modules are used for the host analysis and development phases, as well as for embedded testing on the target hardware.

The components of the StarLite environment include a Modula-2 compiler, a symbolic debugger, a profiler, an interpreter for the architecture, a window package, a simulation package, and a concurrent transaction execution facility. The compiler and interpreter are implemented in C for portability; the rest of the software is in Modula-2. The environment has been used to develop a non-proprietary, UNIX-like operating system that is designed for a multiprocessor architecture, as well as to perform experiments with concurrency control algorithms for real-time database systems. Both systems are organized as module hierarchies composed from reusable components.

As one measure of the effectiveness of the environment, it is often possible to fix errors in the operating system, compile, and reboot the StarLite virtual machine in less than twenty seconds. The total

compilation time on a SUN 3/280 for the 66 modules (7500 lines) that comprise the operating system is 16 seconds. The StarLite interpreter, as measured by Wirth's Modula-2 benchmark program [1], executes at a speed of from one to six times that of a PDP 11/40, depending on the mix of instructions.

Another measure of the effectiveness of the environment is the ease of developing and evaluating application software. Real-time database software, which is one of the target research areas, is being developed to demonstrate StarLite's capability. The growing importance of real-time systems in a large number of applications, such as aerospace and defense systems, industrial automation, and commercial/business applications, has been well acknowledged, and has resulted in an increased research effort in this area [2, 3, 4].

However, evaluating the performance of real-time software or even testing new algorithms has required a large investment of time and resources. One of the primary reasons for the difficulty in successfully evaluating a distributed real-time software is that it takes time to develop a system. Furthermore, evaluation is complicated since it involves a large number of system parameters that may change dynamically. As a result, the field of distributed real-time software evaluation currently lags other research areas. We feel that an important reason for this situation is that many interrelated factors affecting performance (concurrency control, buffering schemes, data distribution, etc.) have been studied as a whole, without completely understanding the overhead imposed by each. An evaluation based on a combination of performance characterization and modeling is necessary in order to understand the impact of control algorithms on the performance of distributed real-time systems.

The StarLite environment can reduce the time and effort necessary for evaluating new technologies and design alternatives for distributed real-time software. Although there exist tools for system development and analysis, few tools exist for distributed real-time system experimentation. Especially if the system designer must deal with message-passing protocols and distributed data, it is essential to have an appropriate environment for success in the design and analysis tasks.

Recently, simulators have been developed for investigating performance of several priority-based concurrency control algorithms for real-time applications [5, 6, 7]. However, they do not provide a module hierarchy composed from reusable components as in our environment. Software developed in the StarLite environment will execute in a given target machine without modification of any layer except the hardware interface. In the operating system area, the ARTS real-time kernel and its toolset, being developed at Carnegie-Mellon University, attempts to provide a "predictable, analyzable, and reliable distributed real-time computing environment" which is an excellent foundation for a real-time system [8]. It implements different prioritized and non-prioritized scheduling algorithms and prioritized message passing. The major difference between our environment from ARTS is that ours is portable since it is implemented in a host environment, and our environment can support a spectrum of distributed database system functions without much overhead.

When developing software systems, there should be assumptions and requirements about the environment and the target system. They form the basis for evaluating tools and environments. In this paper, we first discuss assumptions and requirements for StarLite. We then discuss the database interface of the StarLite environment. One of the benefits of having an environment such as StarLite is that we can develop application software that provides rapid answers to technical questions. To demonstrate the capability of the StarLite environment, real-time database systems have been prototyped. The results of experiments with those systems are described.

2. Assumptions

There are three problems to address when developing software: intrinsic, technological, and software life-cycle problems. Our primary assumption is that the solutions to technological problems compose a relatively small percentage of a typical system's code, even though a large percentage of the design phase may be occupied with technology issues. Thus, the majority of the code deals with intrinsic problems, such as the protection mechanism for a file server, rather than with technology issues, such as the access time or capacity of a disk. By properly isolating the technology-dependent portions of a

software system behind virtual machine interface definitions, software can be developed in a host environment that is separated from the target hardware.

As IBM discovered with their VM system, it is cost-effective to provide environments whose sole purpose is to support software development. A host operating system will always provide a friendlier development environment than a target hardware system. The bare machine environment is the worst possible place in which to explore new software concepts. For example, even the recovery of the event history leading up to an error in a distributed system can be a difficult and, in some cases, an impossible task.

Debugging is greatly facilitated in the host environment. The StarLite symbolic debugger supports the examination of an arbitrary number of execution "threads". As a result, the state of a distributed computation can be examined "as a whole". In addition to aiding fault isolation, the use of a host environment also facilitates fault insertion. For example, the packet error rate on a subnet could be increased to determine the effect on an internet.

Before the use of a host environment becomes feasible, however, it must be possible to simulate the machine-dependent components of a system on the host. The first step towards this goal is to require machine-independent interfaces. For instance, rather than referring to a machine status word at an absolute address, the operating system might invoke a procedure to return the word's value. When the system is executed on the host, the implementation module corresponding to the procedure would simulate the actions of the target hardware. When executing on the target, the implementation would read the content of the absolute address.

Next, for each device used in a project, it is necessary to implement a validated simulation model. This property is necessary for correctness and for the support of performance studies in which the analysis of a prototype is used to predict performance on the target hardware. The results of the virtual machine studies by Canon [9] indicate that these assumptions are reasonable. With StarLite, it is also possible to execute in a "hybrid" mode in which some modules execute on the target and some on the host. For

instance, the disk for a file server could either be a target disk system or a simulated disk when the server code is executing on the host.

It is also possible to capture the timing effects of instruction execution, but not to the level of individual instructions. If that degree of accuracy is required, a VM implementation should be considered.

The final assumption is for the existence of a high-level language whose compiler supports separate compilation of units and generates reentrant code. The StarLite system can be replicated for any such language.

The separate compilation feature is used to "build" a system. For example, the Coroutines module, which is normally implemented in assembly language, forms the basis for the concurrent programming kernel, Processes. The concurrent programming kernel is then used to build the simulation package. Finally, the concurrent programming kernel, the simulation, and window packages are used to implement the virtual machine interface of the application package.

In order to facilitate rapid prototyping, we are developing a library of generic device objects. At present, the object library includes processes, clocks, disks, and Ethernets. Each device is presented to the user as an abstract data type, which is implemented by using the simulation package to model its characteristics and the window package to display its actions. As each data type is instantiated, a window is created to display the operations on that instance and also to serve as a point of interaction with the user. For example, a disk window provides a profile view of the device with a moving pointer to indicate head movement and sector selection. In designing the window interface, the goals were to present uniform options that could be used either in "hybrid" mode for real devices or in host-only mode.

3. Requirements for Integrated Environment

The primary project requirement for StarLite is that software developed in the environment must be capable of being retargeted to different architectures only by recompiling and replacing a few low-level modules. The anticipated benefits are fast prototyping times, greater sharing of software in the research community, and the ability for one research group to validate the claims of another by replicating

experimental conditions exactly.

The StarLite architecture is designed to support the simultaneous execution of multiple operating systems in a single address space. For example, to prototype a distributed operating system, we might want to initiate a file server and several clients. Each virtual machine would have its own operating system and user processes. All of the code and data for all of the virtual machines would be executed as a single UNIX process. In order to support this requirement, we assume the existence of high-performance workstations with large local memories. Ideally, we would prefer multi-thread support, but multiprocessor workstations are not yet widely available. We also assume that hardware details can be isolated behind high-level language interfaces to the extent that the majority of a system's software remains invariant when retargeted from the host to a target architecture.

Figure 1 illustrates the use of the environment during a test session for the StarLite operating system. The figure illustrates our proprietary UNIX implementation "booting up" on a six node virtual network. Once the virtual network has booted, the system designer can can execute test programs, collect statistics, or examine the system state using the builtin debugger, which is illustrated in Figure 2.

The architectural requirements to be satisfied by an interpreter that supports multiple operating systems running in a single, large address space are interesting. They include high speed, compact code, good error detection, demand loading, dynamic restart, fast context switches, hybrid execution modes, and portability.

High Speed. Obviously, the speed of the host architecture is a determining factor in the usefulness of any prototyping effort. Prototyping is most effective for logic-intensive programs, such as operating systems, because the ratio of code to code-executed-per-function is high. For example, running user programs at the shell level on top of the prototype operating system, which is running on an interpreter, provides a response-level comparable (several seconds) to a PDP-11. As the number of users increase or as the number of data-intensive applications increase, the response time increases considerably. Data-intensive programs tend to apply a large percentage of their code to each data point. Thus, the number of

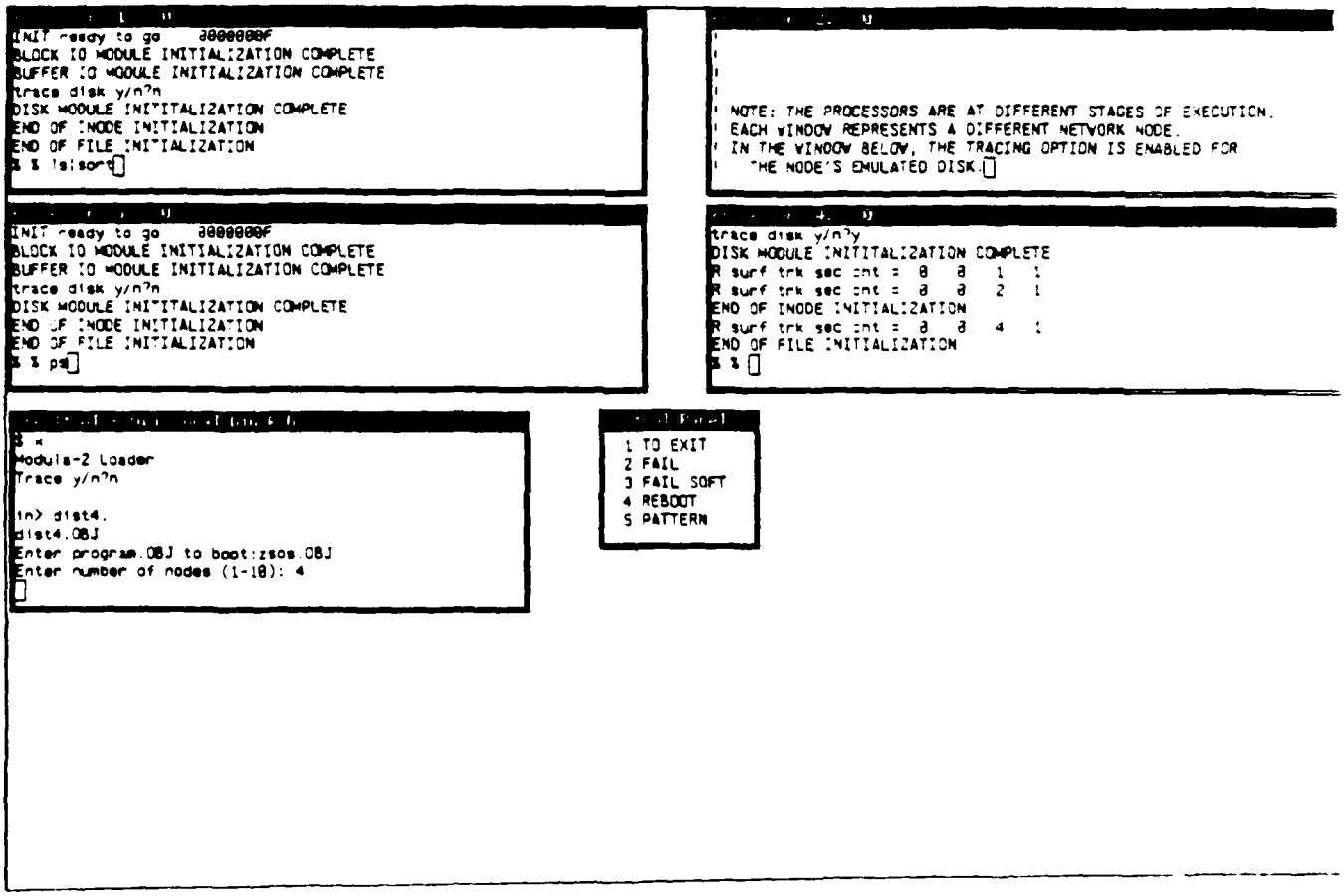


Figure 1. A Six-Node StarLite System Running UNIX

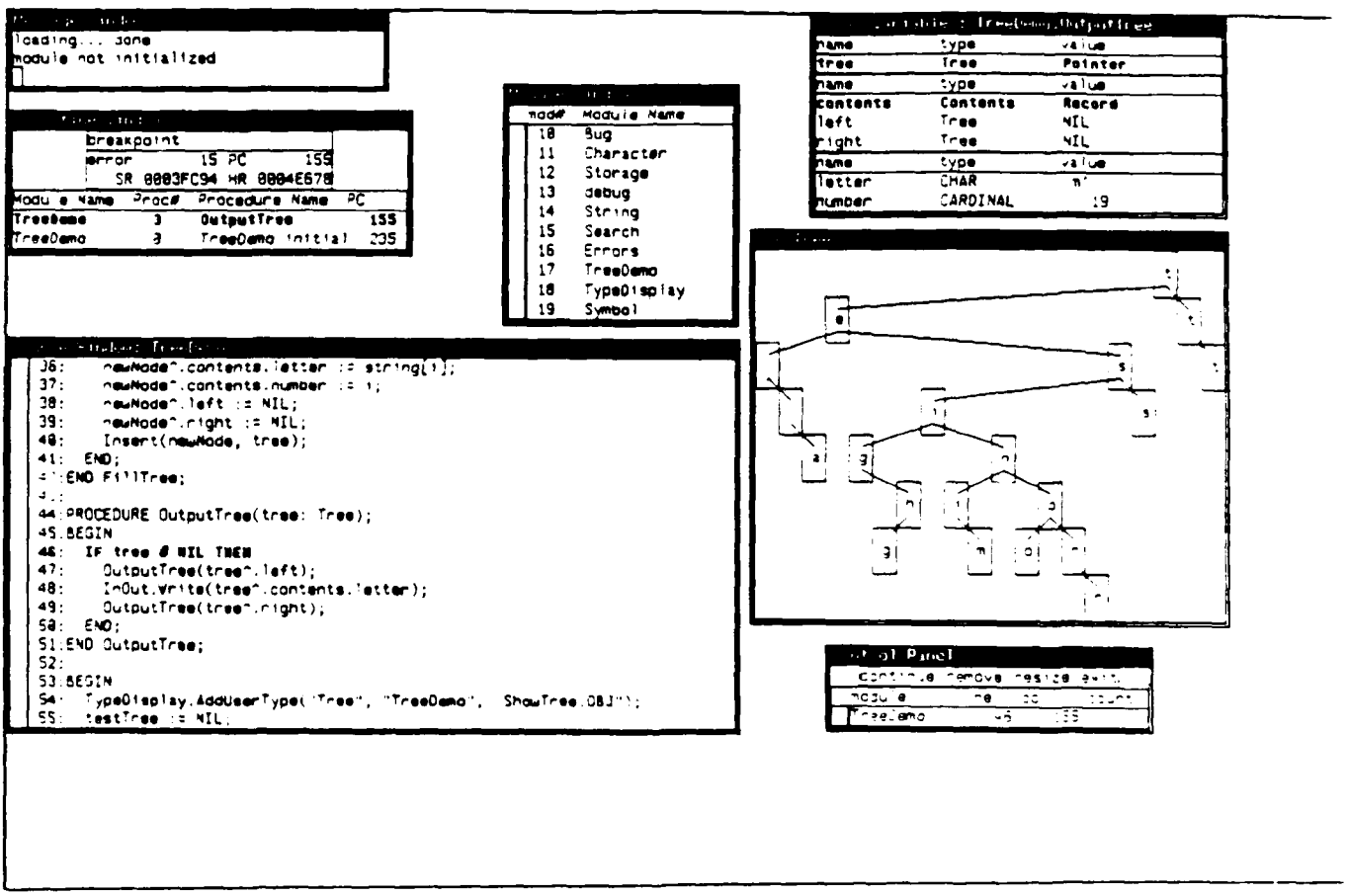


Figure 2. The StarLite Symbolic Debugger

data points determines execution speed. In many cases, having fast machines is the only effective way to prototype data-intensive applications.

Since the StarLite system uses an interpreter to define its virtual machines, we tend to stay away from data-intensive test programs. It would be nice to have an execution speed comparable to a bare machine, but that could only be achieved by building a software prototyping workstation. For now, we are satisfied as long as the edit-compile-boot-and-test cycle is significantly faster than any other environment.

Compact Code. The generated code for the StarLite architecture is extremely space efficient. For example, the object code (.o file) sizes for a sample 1,000 line program were SUN3-Modula2(130K), SUN3-C(65K), PC286-C(35K), StarLite-Modula2(11K). Compact code has a significant effect on the speed with which the environment can load both system components and user-level programs that might run on those components. Compactness also increases cache locality, reduces page faults, and maximizes the quantity of software that can be co-resident in the system.

Error Detection. The benefits of integrity checking as an essential component of a language's implementation have been discussed by Wirth [10]. The StarLite architecture supports checks for overflow/underflow, division by zero, subrange and subscript checking, NIL pointer checks, illegal addresses, and stack overflow. Subrange and subscript checks are generated by the compiler.

Demand Loading. The StarLite architecture supports demand loading; that is, modules are loaded at the point that one of their procedures is called. Thus, a large software system begins execution very quickly and then loads only the modules that are actually referenced. For example, one version of the operating system defers loading the file system, or even the disk driver, until a file operation is performed.

At the current time, a linker is superfluous; as soon as a module is compiled, it may be executed. Demand loading and the absence of linking greatly enhances the efficacy of the StarLite debug cycle. The only limit on debugging is how fast the programmer can discover bugs and type in the changes.

Dynamic Restart. When debugging software, it can be annoying to discover an error, return to the host level, compile, and then run the system to the point of error only to discover another silly mistake. The StarLite architecture is designed so that an IMPLEMENTATION module can be compiled in a child process while the interpreter is suspended. That module can be reinserted into memory and the system restarted.

Another dynamic restart feature supports the emulation of partial failure as might be experienced in a distributed system. The Modula-2 compiler does not attempt to statically initialize any data area. Thus, any module, or set of modules, can be dynamically restarted at any time without reloading the object modules from disk. For a distributed system, the user can induce virtual processor failures and then "bring up" the operating system on those nodes without loading any software from disk.

Fast Context Switches. Unlike the "high-speed" requirement, achieving a fast context switch time can be realized independent of the characteristics of the host machine. For example, there are no context switches within the interpreter, which is basically a C procedure in a closed loop. Therefore, a host architecture with a slow context switch time has no effect on the interpreter's context switch time; it is only a function of the state information that must be saved and restored. This is an important requirement as a typical operating system "run" can involve thousands of context switches.

Each implementation of the architecture must be balanced to match the characteristics of the host machine. The current SUN 3/280 interpreter executes 200,000 coroutine transfers/second. On the other hand, the IBM PS2/50 interpreter executes at 10,000 transfers/second.

Hybrid Execution Modes. In a software development environment, it is advantageous to use services that already exist in the host environment. For example, it is possible to "mount" the host file system on a leaf of a prototyped file system, or even as the prototype's "root" file system. Another example would be to use the host's database services.

The keys to hybrid execution are architectural support and the definition of interfaces that remain invariant to changes in implementation technology. Emulation services are usually implemented by VM

ROM routines. VM ROM can be used to provide functionality that the prototype software does not. A VM ROM routine has a DEFINITION module but its implementation is part of the interpreter. At execution time, the architecture intercepts calls to procedures in VM ROM and directs them to C routines. For example, when prototyping an operating system to experiment with file system issues, it is not necessary to worry about program management; VM ROM routines can be used to interface to an existing file system. At a later stage of development, the VM ROM code can be gradually replaced with code for a prototype file system.

It is easy to add additional packages to the VM ROM interface. The disadvantage is that all ROM packages must be co-resident with the interpreter. In a future version of StarLite under IBM's OS/2, all of the ROM packages will be dynamically linked on demand.

Portability. One of the benefits of developing systems in the StarLite environment is that the code can be shared with other researchers. To facilitate sharing at the object code level, the instructions generated by the compiler and its object module format are canonical. That is, the byte ordering is fixed, as is the character code (ASCII), and the floating point format (IEEE). If the host has different conventions, the compiler performs the conversions as it generates code. To the extent that an implementation module is machine invariant, it should be possible to transmit object modules from one site to another and to have them work.

4. Transaction Management Interface

The transaction management interface of the StarLite environment is designed to facilitate easy extensions and modifications. Server processes can be created, relocated, and new implementations of server processes can be dynamically substituted. It efficiently supports a spectrum of distributed database functions at the operating system level, and facilitates the construction of multiple database systems with different characteristics. For experimentation, system functionality can be adjusted according to application-dependent requirements without much overhead for new system setup. Since one of the design goals of the StarLite system is to conduct an empirical evaluation of the design and

implementation of application software for transaction management, it has built-in support for performance measurement of both elapsed time and blocked time for each transaction [11, 12].

The transaction management module library provides support for concurrent multi-transaction execution, including transparency to concurrent access, data distribution, and atomicity. The environment can manage any number of virtual sites specified by the user. Modules that implement transaction processing are decomposed into several server processes, and they communicate among themselves through ports. The clean interface between server processes simplifies incorporating new algorithms into the environment, or testing alternate implementations of algorithms. To permit concurrent transactions on a single site, there is a separate process for each transaction that coordinates with other server processes. Figure 3 illustrates the structure of the transaction management environment.

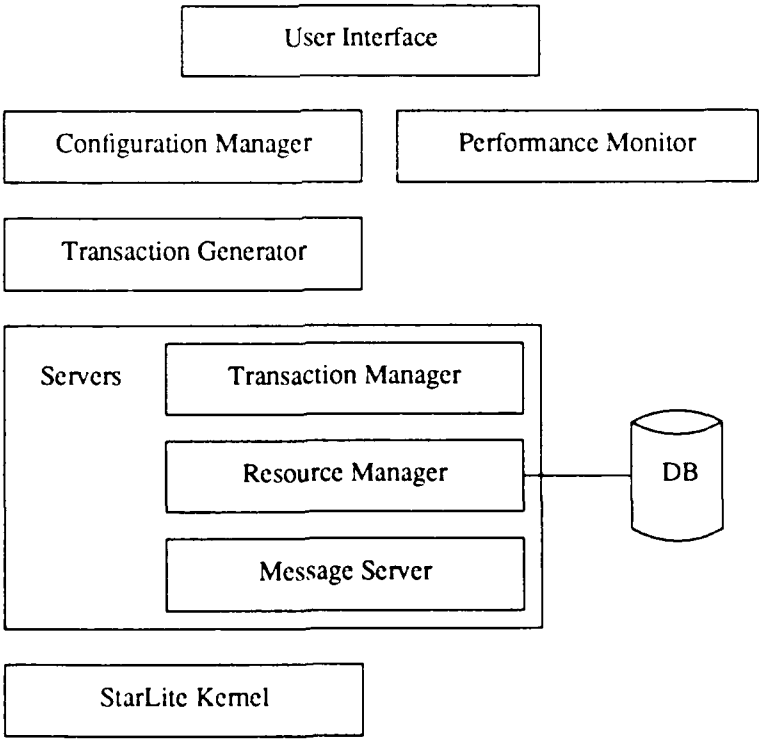


Figure 3. Structure of The Transaction Management Environment

The User Interface (UI) is a front-end invoked when the environment begins execution. UI is menu-driven, and designed to be flexible in allowing users to experiment with various configurations of system parameters. A user can specify the following:

- system configuration: number of sites and topology, and the relative speed of CPU, I/O, and communication cost.
- database configuration: database at each site with user defined structure, size, granularity, and levels of replication.
- load characteristics: number of transactions to be executed, size of their read-sets and write-sets, transaction types (read-only or update) and their priorities, and the mean interarrival time of transactions.
- concurrency control: locking, timestamp ordering, and priority-based.

UI initiates the Configuration Manager (CM) that initializes the data structures necessary for transaction processing from user specifications. CM invokes the Transaction Generator at appropriate time intervals to generate the next transaction to form a Poisson distribution of transaction arrival times. When a transaction is generated, it is assigned an identifier that is unique among all transactions in the system.

Transaction execution consists of read and write operations. Each read or write operation is preceded by an access request sent to the Resource Manager, which maintains the local database at each site. If the access request cannot be granted, the Transaction Manager (TM) executes either a blocking operation to wait until the data object can be accessed, or an abort procedure, depending on the situation. Transactions commit in two phases. The first commit phase consists of at least one round of messages to determine if the transaction can be globally committed. Additional rounds may be used to handle potential failures. The second commit phase causes the data objects to be written to the database for successful transactions. TM executes the two commit phases to ensure that a transaction commits or aborts globally.

The Message Server (MS) is a process listening on a well-known port for messages from remote sites. When a message is sent to a remote site, it is placed on the message queue of the destination site

and the sender blocks itself on a private semaphore until the message is retrieved by MS. If the receiving site is not operational, a time-out mechanism will unblock the sender process. When MS retrieves a message, it wakes the sender process and forwards the message to the proper servers or TM. The environment implements Ada-style rendezvous (synchronous) as well as asynchronous message passing. Inter-process communication within a site does not go through the Message Server; processes send and receive messages directly through their associated ports.

The inter-process communication structure is designed to provide a simple and flexible interface to the client processes of the application software, independent from the low-level hardware configurations. It is split into three levels of hierarchy: transport layer, network layer, and physical layer.

The Transport layer is the interface to the application software, thus it is designed to be as abstract as possible in order to support different port structures and various message types. In addition, application level processes need not know the details of the destination device. The invariant built into the design of the inter-process communication interface *is that the application level sender allocates the space for a* message, and the receiver deallocates it. Thus, it is irrelevant whether or not the sender and receiver share memory space, i.e., whether or not the Physical layer on the sender's side copies the message into a buffer and deallocates it at the sender's site, and the Physical layer at the receiver's site allocates space for the message. This enables prototyping distributed systems or multiprocessors with no shared memory, as well as multiprocessors with shared memory space. When prototyping the latter, only addresses need to be passed in messages without intermediate allocation and deallocation.

The Physical layer of message passing simulates the physical sending and receiving of bits over a communication medium, i.e., it is for intersite message passing. The device number in the interface is simply a cardinal number; this enables the implementation to be simple and extensible enough to support any application. To simulate sending or to actually send over an Ethernet in the target system, for example, a module could map network addresses onto cardinals. To send from one processor to another in a multiprocessor or distributed system, the cardinals can represent processor numbers.

Messages are passed to specific processes at specific sites in the Network layer of the communications interface. This layer serves to separate the Transport and the Physical layers, so that the Transport layer interface can be processor- and process-independent and the Physical layer interface need be concerned only with the sending of bits from one site to another. The Transport layer interface of the communication subsystem is implemented in the Transport module. A Transport-level Send is made to an abstraction called a *PortTag*. This abstraction is advantageous because the implementation (i.e., what a PortTag represents) is hidden in the Ports module. Thus the PortTag can be mapped onto any port structure or the reception points of any other message passing system. The Transport-level Send operation builds a packet consisting of the sender's PortTag, used for replies, the destination PortTag, and the address of the message. It then retrieves from the destination PortTag the destination device number. If this number is the same as the sender's, the Send is an intra-site message communication, and hence the middle-level Send is performed. Otherwise the send requires the Physical module for intersite communication. Note that accesses to the implementation details of the PortTag are restricted to the module that actually implements it; this enables changing the implementation without recompiling the rest of the system.

The Performance Monitor interacts with the transaction managers to record, priority/timestamp and read/write data set for each transaction, time when each event occurred, statistics for each transaction and cpu hold interval in each node. The statistics for a transaction includes arrival time, start time, total processing time, blocked interval, whether deadline was missed or not, and number of aborts.

5. A Real-Time Database Experiment

The previous section described the structure of the transaction management of the StarLite environment. In this section, we present a real-time database system developed using the environment. Two goals of our work were 1) evaluation of the environment itself in terms of correctness, functionality, and modularity, by using it in implementing a distributed real-time database system, and 2) performance comparison between two-phase locking and priority-based locking algorithms through a sensitivity study of

key parameters that affect performance.

5.1. Priority-Based Synchronization

It has been recognized that database systems are assuming much greater importance in real-time systems that require high reliability, high performance, and predictability [3, 6, 8]. State-of-the-art database systems are typically not used in real-time applications due to two inadequacies: poor performance and lack of predictability.

In a real-time database system, synchronization protocols must not only maintain the consistency constraints of the database but also satisfy the timing requirements of the transactions accessing the database. To satisfy both the consistency and real-time constraints, there is a need to integrate synchronization protocols with real-time priority scheduling protocols [13]. A major source of problems in integrating the two protocols is the lack of coordination in the development of synchronization protocols and real-time priority scheduling protocols. Due to the effect of blocking in lock-based synchronization protocols, a direct application of a real-time scheduling algorithm to transactions may result in a condition known as *priority inversion.*

Priority inversion is said to occur when a higher priority process is forced to wait for the execution of a lower priority process for an indefinite period of time. When the transactions of two processes attempt to access the same data object, the access must be serialized to maintain consistency. If the transaction of the higher priority process gains access first, then the proper priority order is maintained; however, if the transaction of the lower priority gains access first and then the higher priority transaction requests access to the data object, this higher priority process will be blocked until the lower priority transaction completes its access to the data object. Priority inversion is inevitable in transaction systems. However, to achieve a high degree of schedulability in real-time applications, priority inversion must be minimized. This is illustrated by the following example.

Example: Suppose T_1, T_2, and T_3 are three transactions arranged in descending order of priority with T_1 having the highest priority. Assume that T_1 and T_3 access the same data object O_1. Suppose that

at time t_1 transaction T_3 obtains a lock on O_i. During the execution of T_3, the high priority transaction T_1 arrives, preempts T_3 and later attempts to access the object O_i. Transaction T_1 will be blocked, since O_i is already locked. We would expect that T_1, being the highest priority transaction, will be blocked no longer than the time for transaction T_3 to complete and unlock O_i. However, the duration of blocking may, in fact, be unpredictable. This is because transaction T_3 can be blocked by the intermediate priority transaction T_2 that does not need to access O_i. The blocking of T_3, and hence that of T_1, will continue until T_2 and any other pending intermediate priority level transactions are completed.

The blocking duration in the example above can be arbitrarily long. This situation can be partially remedied if transactions are not allowed to be preempted; however, this solution is only appropriate for very short transactions, because it creates unnecessary blocking. For instance, once a long low priority transaction starts execution, a high priority transaction not requiring access to the same set of data objects may be needlessly blocked.

An approach to this problem, based on the notion of *priority inheritance*, has been proposed [14]. The basic idea of priority inheritance is that when a transaction T of a process blocks higher priority processes, it executes at the highest priority of all the transactions blocked by T. This simple idea of priority inheritance reduces the blocking time of a higher priority transaction. However, this is inadequate because the blocking duration for a transaction, though bounded, can still be substantial due to the potential *chain of blocking*. For instance, suppose that transaction T_1 needs to sequentially access objects O_1 and O_2. Also suppose that T_2 preempts T_3 which has already locked O_2. Then, T_2 locks O_1. Transaction T_1 arrives at this instant and finds that the objects O_1 and O_2 have been respectively locked by the lower priority transactions T_2 and T_3. As a result, T_1 would be blocked for the duration of two transactions, once to wait for T_2 to release O_1 and again to wait for T_3 to release O_2. Thus a chain of blocking can be formed.

One idea for dealing with this inadequacy is to use a total priority ordering of active transactions [15]. A transaction is said to be *active* if it has started but not yet completed its execution. A transaction

can be active in one of two states: executing or being preempted in the middle of its execution. The idea of total priority ordering is that the real-time locking protocol ensures that each active transaction is executed at some priority level, taking priority inheritance and read/write semantics into consideration.

5.2. Experiments with Priority Ceiling

To ensure the total priority ordering of active transactions, three priority ceilings are defined for each data object in the database: the write-priority ceiling, the absolute-priority ceiling, and the rw-priority ceiling. The write-priority ceiling of a data object is defined as the priority of the highest priority transaction that may write into this object, and absolute-priority ceiling is defined as the priority of the highest priority transaction that may read or write the data object. The rw-priority ceiling is set dynamically. When a data object is write-locked, the rw-priority ceiling of this data object is defined to be equal to the absolute priority ceiling. When it is read-locked, the rw-priority ceiling of this data object is defined to be equal to the write-priority ceiling.

The priority ceiling protocol is premised on systems with a fixed priority scheme. The protocol consists of two mechanisms: *priority inheritance* and *priority ceiling*. With the combination of these two mechanisms, we get the properties of freedom from deadlock and a worst case blocking of at most a single lower priority transaction.

When a transaction attempts to lock a data object, the transaction's priority is compared with the highest rw-priority ceiling of all data objects currently locked by other transactions. If the priority of the transaction is not higher than the rw-priority ceiling, the access request will be denied, and the transaction will be blocked. In this case, the transaction is said to be blocked by the transaction which holds the lock on the data object of the highest rw-priority ceiling. Otherwise, it is granted the lock. In the denied case, the priority inheritance is performed in order to overcome the problem of uncontrolled priority inversion. For example, if transaction T blocks higher transactions, T inherits P_H, the highest priority of the transactions blocked by T.

Under this protocol, it is not necessary to check for the possibility of read-write conflicts. For instance, when a data object is write-locked by a transaction, the rw-priority ceiling is equal to the highest priority transaction that can access it. Hence, the protocol will block a higher priority transaction that may write or read it. On the other hand, when the data object is read-locked, the rw-priority ceiling is equal to the highest priority transaction that may write it. Hence, a transaction that attempts to write it will have a priority no higher than the rw-priority ceiling and will be blocked. Only the transaction that read it and have priority higher than the rw-priority ceiling will be allowed to read-lock it, since read-locks are compatible. Using the priority ceiling protocol, mutual deadlock of transactions cannot occur and each transaction can be blocked by at most by one lower priority transactions until it completes or suspends itself.

The total priority ordering of active transactions leads to some interesting behavior. As shown in the example above, the priority ceiling protocol may forbid a transaction from locking an unlocked data object. At first sight, this seems to introduce unnecessary blocking. However, this can be considered as the "insurance premium" for preventing deadlock and achieving block-at-most-once property.

Using the StarLite environment, we have investigated issues associated with this idea of total ordering in priority-based scheduling protocols. One of the critical issues related to the total ordering approach is its performance compared with other design alternatives. In other words, it is important to figure out what is the actual cost for the "insurance premium" of the total priority ordering approach.

In our experiments, all transactions are assumed to be *hard* in the sense that there will be no value in completing a transaction after its deadline. Transactions that miss the deadline are aborted, and disappear from the system. We have used transaction size (the number of data objects a transaction needs to access) as one of the key variables in the experiments. It varies from a small fraction up to a relatively large portion (10%) of the database so that conflicts would occur frequently. The high conflict rate allows synchronization protocols to play a significant role in determining system performance. We chose the arrival rate so that protocols are tested in a heavily loaded rather than lightly loaded system. For designing real-time systems, one must consider high load situations. Even though they may not arise frequently, one

would like to have a system that misses as few deadlines as possible when such peaks occur. In other words, when a crisis occurs and the database system is under pressure is precisely when making a few extra deadlines could be most important [16]. Due to space considerations, we summarize our findings briefly to illustrate the performance of the algorithms.

As the transaction size increases, there is little impact on the throughput of the priority-ceiling protocol over a range of transaction sizes and over various workload. This is because in the priority-ceiling protocol, the conflict rate is determined by ceiling blocking rather than direct blocking, and the frequency of ceiling blocking is not sensitive to the transaction size. However, the performance of the two-phase locking protocol with or without priority degrades very rapidly. This phenomenon is more pronounced as the transaction workload becomes more I/O bound, since there are few conflicts for the small transactions in the two-phase locking protocol, and the concurrency is fully achieved with an assumption of parallel I/O processing. Poor performance of the two-phase locking protocol for bigger transactions is due to the high conflict rate.

Another important performance statistic is the percentage of deadlines missed by transactions, since the synchronization protocol in real-time database systems must satisfy the timing constraints of individual transactions. In our experiments, each transaction's deadline is set in proportion to its size and system workload (number of transactions), and the transaction with the earliest deadline is assigned the highest priority. The percentage of deadlines missed by transactions increases sharply for the two-phase locking protocol as the transaction size increases. A sharp rise was expected, since the probability of deadlocks would go up with the fourth power of the transaction size [17]. However, the percentage of deadlines missed by transactions increases more slowly as the transaction size increases in the priority-ceiling protocol. Since there is no deadlock in the priority-ceiling protocol, the response time is proportional to the transaction size and the priority ranking.

6. Conclusions

We cannot offer proof that the StarLite system is the appropriate environment for distributed real-time systems research. However, we have shown that it is feasible to pursue major systems projects in a virtual interface environment. Even though the environment cannot execute programs as fast as a physical machine and it would be infeasible to emulate all of a physical machine's effects, such as memory interference, the advantages are a greatly accelerated development cycle and totally portable, and hence reproducible, results.

Although the complexity of a distributed real-time software makes integrated environments difficult to develop, the implementation has proven satisfactory for experimentation of design choices, different database and operating system techniques, and even an integrated evaluation of real-time systems. It supports a very flexible user interface to allow a wide range of system configurations and workload characteristics. Since the StarLite environment is designed to provide a spectrum of database functions and operating system modules, it facilitates the construction of multiple system instances with different characteristics without much overhead. Expressive power and performance evaluation capability of our environment has been demonstrated by implementing a real-time database system and investigating the performance characteristics of the priority-ceiling protocol.

StarLite is possible because workstations now have large physical memories and are fast enough to run interpreters at the speed of physical machines ten years ago. Ten years ago it would not have been feasible to run an emulator on a PDP-11 and then to implement a database system on top of it. StarLite does not currently take advantage of the multi-thread support available on some of the newer workstations, but it could.

References

[1] Wirth, N., *Programming in Modula-2*, Springer-Verlag, (1983).

[2] Buchmann, A. et al., "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control," *Fifth Data Engineering Conference*, Feb. 1989, 470-480.

[3] Son, S. H., "Real-Time Database Systems: Issues and Approaches," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, (March 1988).

[4] Son, S. H. and H. Kang, "Approaches to Design of Real-Time Database Systems," *International Symposium on Database Systems for Advanced Applications*, Seoul, Korea, (April 1989), 274-281.

[5] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Study," *VLDB Conference*, Sept. 1988, pp 1-12.

[6] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *VLDB Conference*, August 1989.

[7] Rajkumar, R., "Task Synchronization in Real-Time Systems," *Ph.D. Dissertation*, Carnegie-Mellon University, August 1989.

[8] Tokuda, H. and C. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM Operating Systems Review 23*, 3, July 1989.

[9] Canon, M.D. et al, "A Virtual Machine Emulator for Performance Evaluation," *Communications of the ACM 23*, 2 (Feb. 1980), 71-80.

[10] Wirth, N., "Microprocessor Architectures: A Comparison Based on Code Generation by Compiler," *Communications of the ACM 29*, 10 (Oct. 1986), 978-994.

[11] Son, S. H., "A Message-Based Approach to Distributed Database Prototyping," *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, May 1988, 71-74.

[12] Son, S. H. and Y. Kim, "A Software Prototyping Environment and Its Use in Developing a Multiversion Distributed Database System," *18th International Conference on Parallel Processing*, St. Charles, Illinois, August 1989, Vol. 2. 81-88.

[13] Son, S. H., "On Priority-Based Synchronization Protocols for Distributed Real-Time Database Systems," *IFAC/IFIP Workshop on Distributed Databases in Real-Time Control*, Budapest, Hungary, October 1989, 67-72.

[14] Sha, L., R. Rajkumar, and J. Lehoczky, Priority Inheritance Protocol: An Approach to Real-Time Synchronization, *IEEE Transaction on Computers* (to appear).

[15] Sha, L., R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988, 82-98.

[16] Son, S. H. and C. Chang, "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment," *10th International Conference on Distributed Computing Systems*, Paris, France, June 1990.

[17] Gray, J. et al., "A Straw Man Analysis of Probability of Waiting and Deadlock," *IBM Research Report*, RJ 3066, 1981.

Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment

Sang H. Son* and Chun-Hyon Chang**

* Department of Computer Science
University of Virginia
Charlottesville, VA 22903, USA

** Department of Computer Science
Kon-Kuk University
Seoul, Korea

ABSTRACT

Real-time systems must maintain data consistency while minimizing the number of tasks that miss the deadline. To satisfy both the consistency and real-time constraints, there is the need to integrate synchronization protocols with real-time priority scheduling protocols. In this paper, we address the problem of priority scheduling in real-time database systems. We first present a prototyping environment for investigating distributed software. Specific priority-based real-time locking protocols are then discussed, together with a performance study which illustrates the use of the prototyping environment for evaluation of synchronization protocols for real-time database systems.

1. Introduction

The growing importance of real-time computing in a wide range of applications such as aerospace and defense systems, industrial automation, and nuclear reactor control, has resulted in an increased research effort in this area. Distributed systems greatly exacerbate the difficulty of developing real-time systems as delays associated with interprocess communications and remote database accesses must be taken into account [Wat88]. Researchers working on developing real-time systems based on distributed system architecture have found out that database managers are assuming much greater importance in real-time systems. In the recent workshops sponsored by the Office of Naval Research [IEEE89, ONR89], developers of real-time systems pointed to the need for basic research in database systems that satisfy timing constraint requirements in collecting, updating, and retrieving shared data. Further evidence of its importance is the recent growth of research in this field [Shin87, Son88b].

Compared with traditional databases, real-time database systems have a distinct feature: they must satisfy the timing constraints associated with transactions. In other words, "time" is one of the key factors to be considered in real-time database systems. The timing constraints of a transaction typically include its ready time and deadline, as well as temporal consistency of the data accessed by it. Transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query on a tracking data of a missile must be processed within the given deadlines: otherwise, the information provided could be of little value. In such a system, transaction processing must satisfy not only the database consistency constraints but also the timing constraints.

In addition to providing shared data access capabilities, distributed real-time database systems offer a means of loosely coupling communicating processes, making it easier to rapidly update software, at least from a functional perspective. However, with respect to time-driven scheduling and system timing predictability, they present new problems. One of the characteristics of current database managers is that they do not schedule their transactions to meet response time requirements and they commonly lock data tables to assure database consistency. Locks and time-driven scheduling are basically incompatible. Low priority transactions holding locks required by higher priority transactions can and will block the higher priority transactions, leading to response requirement failures. New techniques are required to manage data consistency which are compatible with time-driven scheduling.

One of the primary reasons for the difficulty in successfully developing and evaluating new database management techniques suitable for real-time applications is that it takes a long time to develop a system, and evaluation is complicated because it involves a large number of system parameters that may change dynamically. For example, although new approaches for synchronization and database recovery have been developed recently [Son88, Son89], experimentation to verify their properties and to evaluate their performance has not been performed due to the lack of appropriate test tools.

A prototyping technique can be applied effectively to the evaluation of database management techniques for distributed real-time systems. A *database prototyping environment* is a software package that supports the evaluation of database management techniques in an environment other than that of the target database system. The advantages of such an environment are obvious [Son90]. Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation, especially for distributed real-

This work was supported in part by ONR contract # N00014-88-K-0245, by DOE grant # DE-FG05-88ER25063, and by IBM FSD under University Agreement WG-249153.

time database systems. Recently, simulators have been developed for investigating performance of several priority-based concurrency control algorithms for real-time applications [Abb88, Abb89, Raj89]. However, they do not provide a module hierarchy composed from reusable components as in our prototyping environment. Software developed in our prototyping environment will execute in a given target machine without modification of any layer except the hardware interface. In the operating system area, the ARTS real-time kernel and its toolset, being developed at Carnegie-Melon University, attempts to provide a "predictable, analyzable, and reliable distributed real-time computing environment" which is an excellent foundation for a real-time system [Tok89]. It implements different prioritized and non-prioritized scheduling algorithms and prioritized message passing. The major difference between our prototyping environment from ARTS is that ours is portable since it is implemented in a host environment, and our environment can support a spectrum of distributed database system functions without much overhead.

This paper presents a database prototyping environment that supports evaluation of distributed real-time database systems. To illustrate its usefulness, a series of experimentation to evaluate priority-based real-time locking protocols has been performed.

One of the major problems in priority-based locking protocols is that, owing to the effect of blocking, a condition known as unbounded *priority inversion*, where a higher priority task is blocked by lower priority tasks for an indefinite period of time. To address this problem, the priority ceiling protocol was proposed in [Sha88]. It tries to achieve not only minimizing the blocking time of a transaction to at most one lower priority transaction execution time, but also preventing the formation of deadlocks. In this paper, we investigate the performance of the priority ceiling protocol and compare it with other synchronization protocols. We also discuss the performance of real-time locking protocols in distributed database environments.

The rest of the paper is organized as follows. Section 2 presents the design principles and the current implementation of the prototyping environment. Section 3 discusses priority-based real-time locking protocols and presents their experimental performance results using the prototyping environment. Section 4 presents two different priority-based locking protocols and their performance in distributed environments. Section 5 is the conclusion.

2. Structure of the Prototyping Environment

A prototyping environment, if properly structured, can reduce the time for evaluating new technologies and design alternatives. From our past experience, we assume that a relatively small portion of a typical database system's code is affected by changes in specific control mechanisms, while the majority of code deals with intrinsic problems, such as file management. Thus, by properly isolating technology-dependent portions of a database system using modular programming techniques, we can implement and evaluate design alternatives very rapidly. For this reason, our prototyping environment is designed as a modular, message-passing system to support easy extensions and modifications. Server processes can be created, relocated, and new implementations of server processes can be dynamically substituted. It provides a library of modules with different performance and reliability characteristics for an operating system as well as database management functions [Son88c, Son90]. Operating system facilities are included in the library because the correct functioning and timing behavior of database control algorithms depends on the appropriate support of the underlying operating system. For experimentation, the module library facilitates the construction of multiple system instances customized according to application-dependent requirements without much overhead.

The prototyping environment provides support for transaction processing, including transparency to concurrent access, data distribution, and atomicity. An instance of the prototyping environment can manage any number of virtual sites specified by the user. Modules that implement transaction processing are decomposed into several server processes, and they communicate among themselves through ports. The clean interface between server processes simplifies incorporating new algorithms and facilities into the prototyping environment, or testing alternate implementations of algorithms. A separate process for each transaction is created for concurrent execution of transactions.

Figure 1 illustrates the structure of the prototyping environment. The prototyping environment is based on a concurrent programming kernel, called the StarLite kernel, which supports process control to create, ready, block, and terminate processes. Based on the StarLite kernel, the environment consists of the modules for user interface, configuration management and transaction generation, transaction manager, message server, resource manager, and performance monitor.

User Interface (UI) is a front-end invoked when the prototyping environment begins. UI is menu-driven, and designed to be flexible in allowing users to experiment various configurations with different system parameters. A user can specify the following:

- system configuration: number of sites and topology, and the relative speed of CPU, I/O, and communication cost.

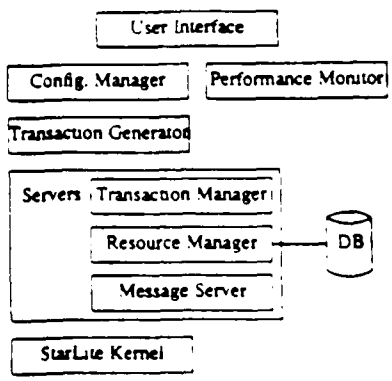


Fig. 1. Structure of the prototyping environment

- database configuration: database at each site with user defined structure, size, granularity, and levels of replication.
- load characteristics: number of transactions to be executed, size of their read-sets and write-sets, transaction types (read-only/update and periodic/aperiodic) and their priorities, and the mean interarrival time of aperiodic transactions.
- concurrency control: locking, timestamp ordering, and priority-based.

UI initiates the Configuration Manager (CM) which initializes necessary data structures for transaction processing based on user specification. CM invokes the Transaction Generator at an appropriate time interval to generate the next transaction.

Transaction execution consists of read and write operations. Each read or write operation is preceded by an access request sent to the Resource Manager, which maintains the local database at each site. Each transaction is assigned to the Transaction Manager (TM). The TM issues service requests on behalf of the transaction and reacts appropriately to the request replies. For instance, if a transaction requests access to a data object that is already locked, the TM executes either blocking operation to wait until the data object can be accessed, or aborting the transaction, depending on the situation. TM executes the two-phase commit protocol to ensure that a transaction commits or aborts globally.

The prototyping environment is currently implemented on a single host. The distributed environment is simulated by the Message Server (MS) listening on a well-known port for messages from remote sites. When a message is sent to a remote site, it is placed on the message queue of the destination site and the sender can block itself on a private semaphore until the message is retrieved by the MS at the receiving site. If the receiving site is not operational, a time-out mechanism will unblock the sender process. When the MS retrieves a message, it wakes the sender process and forwards the message to the proper servers or TM. The prototyping environment implements Ada-style rendezvous (synchronous) as well as asynchronous message passing. Inter-process communication within a site does not go through the Message Server; processes send and receive messages directly through their associated ports.

The Performance Monitor interacts with the transaction managers to record priority/timestamp and read/write data set for each transaction, time when each event occurred, statistics for each transaction in each node. The statistics for a transaction includes arrival time, start time, total processing time, blocked interval, whether deadline was missed or not, and the number of aborts.

3. Prototyping Real-Time Database Systems

In this section, we present a real-time database system prototyped using the prototyping environment. Two goals of our prototyping work are 1) evaluation of the prototyping environment itself in terms of correctness, functionality, and modularity, by using it in prototyping distributed database systems, and 2) performance evaluation of real-time locking and priority-based synchronization protocols through the sensitivity study of key parameters that affect performance.

Real-time databases are often used by applications such as tracking. Tasks in such applications consist of both computing (signal processing) and database accessing (transactions). A task can have multiple transactions, which consists of a sequence of read and write operations operating on the database. Each transaction will follow the two-phase locking protocol [Esw76], which requires a transaction to acquire all the locks before it releases any lock. Once a transaction releases a lock, it cannot acquire any new lock. A high priority task will preempt the execution of lower priority tasks unless it is blocked by the locking protocol at the database. In this section we consider them in a single site environment. Real-time locking protocols in distributed environment is discussed in the next section.

3.1. Priority-Based Synchronization

In a real-time database system, synchronization protocols must not only maintain the consistency constraints of the database but also satisfy the timing requirements of the transactions accessing the database. To satisfy both the consistency and real-time constraints, it is necessary to integrate synchronization protocols with real-time priority scheduling protocols. Due to the effect of blocking in lock-based synchronization protocols, a direct application of a real-time scheduling algorithm to transactions may result in a condition known as *priority inversion*. Priority inversion is said to occur when a higher priority task is forced to wait for the execution of a lower priority task for an indefinite period of time. When two transactions attempt to access the same data object, the access must be serialized to maintain consistency. If the transaction of the higher priority task gains access first, then the proper priority order is maintained; however, if the transaction of the lower priority gains access first and then the higher priority transaction requests access to the data object, this higher priority task will be blocked until the lower priority transaction completes its access to the data object. Priority inversion is inevitable in transaction systems. However, to achieve a high degree of schedulability in real-time applications, priority inversion must be minimized. This is illustrated by the following example.

Example: Suppose T_1, T_2, and T_3 are three transactions arranged in descending order of priority with T_1 having the highest priority. Assume that T_1 and T_3 access the same data object O_i. Suppose that at time t_1 transaction T_3 obtains a lock on O_i. During the execution of T_3, the high priority transaction T_1 arrives, preempts T_3, and later attempts to access the object O_i. Transaction T_1 will be blocked, since O_i is already locked. We would expect that T_1, being the highest priority transaction, will be blocked no longer than the time for transaction T_3 to complete and unlock O_i. However, the duration of blocking may, in fact, be unpredictable. This is because transaction T_3 can be blocked by the intermediate priority transaction T_2 that does not need to access O_i. The blocking of T_3, and hence that of T_1, will continue until T_2 and any other pending intermediate priority level transactions are completed.

The blocking duration in the example above can be arbitrarily long. This situation can be partially remedied if transactions are not allowed to be preempted; however, this solution is only appropriate for very short transactions, because it creates unnecessary blocking. For instance, once a long low priority transaction starts execution, a high priority transaction not requiring access to the same set of data objects may be needlessly blocked.

An approach to this problem, based on the notion of *priority inheritance*, has been proposed [Sha87]. The basic idea of priority inheritance is that when a transaction T of a task blocks higher priority tasks, it executes at the highest priority of all the transactions blocked by T. This simple idea of priority inheritance reduces the blocking time of a higher priority transaction. However, this is inadequate because the blocking duration for a transaction, though bounded, can still be substantial due to the potential *chain of blocking*. For instance, suppose that transaction T_1 needs to sequentially access objects O_1 and O_2. Also suppose that T_2 preempts T_3 which has already locked O_2. Then, T_2 locks O_1. Transaction T_1 arrives at this instant and finds that the objects O_1 and O_2 have been respectively locked by the lower priority transactions T_2 and T_3. As a result, T_1 would be blocked for the duration of two transactions, once to wait for T_2 to release O_1 and again to wait for T_3 to release O_2. Thus a chain of blocking can be formed.

One idea for dealing with this inadequacy is to use a total priority ordering of active transactions [Sha88]. A transaction is said to be *active* if it has started but not yet completed its execution. A transaction can be active in one of two states: executing or being preempted in the middle of its execution. The idea of total priority ordering is that the real-time locking protocol ensures that each active transaction is executed at some priority level, taking priority inheritance and read/write semantics into consideration.

3.2. Total Ordering by Priority Ceiling

To ensure the total priority ordering of active transactions, three priority ceilings are defined for each data object in the database: the write-priority ceiling, the absolute-priority ceiling, and the rw-priority ceiling. The write-priority ceiling of a data object is defined as the priority of the highest priority transaction that may write into this object, and absolute-priority ceiling is defined as the priority of the highest priority transaction that may read or write the data object. The rw-priority ceiling is set dynamically. When a data object is write-locked, the rw-priority ceiling of this data object is defined to be equal to the absolute priority ceiling. When it is read-locked, the rw-priority ceiling of this data object is defined to be equal to the write-priority ceiling.

The priority ceiling protocol is premised on systems with a fixed priority scheme. The protocol consists of two mechanisms: *priority inheritance* and *priority ceiling*. With the combination of these two mechanisms, we get the properties of freedom from deadlock and a worst case blocking of at most a single lower priority transaction.

When a transaction attempts to lock a data object, the transaction's priority is compared with the highest rw-priority ceiling of all data objects currently locked by other transactions. If the priority of the transaction is not higher than the rw-priority ceiling, the access request will be denied, and the transaction will be blocked. In this case, the transaction is said to be blocked by the transaction which holds the lock on the data object of the highest rw-priority ceiling. Otherwise, it is granted the lock. In the denied case, the priority inheritance is performed in order to overcome the problem of uncontrolled priority inversion. For example, if transaction T blocks higher transactions, T inherits P_H, the highest priority of the transactions blocked by T.

Under this protocol, it is not necessary to check for the possibility of read-write conflicts. For instance, when a data object is write-locked by a transaction, the rw-priority ceiling is equal to the highest priority transaction that can access it. Hence, the protocol will block a higher priority transaction that may write or read it. On the other hand, when the data object is read-locked, the rw-priority ceiling is equal to the highest priority transaction that may write it. Hence, a transaction that attempts to write it will have a priority no higher than the rw-priority ceiling and will be blocked. Only the transaction that read it and have priority higher than the rw-priority ceiling will be allowed to read-lock it, since read-locks are compatible. Using the priority ceiling protocol, mutual deadlock of transactions cannot occur and each transaction can be blocked by at most by one lower priority transactions until it completes or suspends itself. For a more formal discussion on the protocol, readers are referred to [Sha88]. The next example shows how transactions are scheduled under the priority ceiling protocol.

Example: Consider the same situation as in the previous example. According to the protocol, the priority ceiling of O_i is the priority of T_1. When T_2 tries to access a data object, it is blocked because its priority is not higher than the priority ceiling of O_i. Therefore T_1 will be blocked only once by T_3 to access O_i, regardless of the number of data objects it may access.

The total priority ordering of active transactions leads to some interesting behavior. As shown in the example above, the priority ceiling protocol may forbid a transaction from locking an unlocked data object. At first sight, this seems to introduce unnecessary blocking. However, this can be considered as the "insurance premium" for preventing deadlock and achieving block-at-most-once property.

Using the prototyping environment, we have been investigating issues associated with this idea of total ordering in priority-based scheduling protocols. One of the critical issues related to the total ordering approach is its performance compared with other design alternatives. In other words, it is important to figure out what is the actual cost for the "insurance premium" of the total priority ordering approach. In our experiments, all transactions are assumed to be *hard* in the sense that there will be no value in completing a transaction after its deadline. Transactions that miss the deadline are aborted, and disappear from the system.

3.3. Performance Evaluation

Various statistics have been collected during the experiments for comparing the performance of the priority ceiling protocol with other synchronization control algorithms. Transaction throughput and the percentage of deadline missing transactions are the most important performance measures in real-time database systems. This section presents these performance measures in a single site database system. Performance in distributed environments will be discussed in the next section.

Transaction are generated with exponentially distributed interarrival times, and the data objects updated by a transaction are chosen uniformly from the database. The total processing time of a transaction is directly related to the number of data objects accessed. Due to space considerations, we do not present all our results but have selected the graphs which best illustrate the difference and performance of the algorithms. For example, we have omitted the results of an experiment that

varied the size of the database, and thus the probability of conflicts, because they only confirm and not increase the knowledge yielded by other experiments.

For each experiment and for each algorithm tested, we collected performance statistics and averaged over the 10 runs. The percentage of deadline-missing transactions is calculated with the following equation: *%missed* = 100 * (number of deadline-missing transactions / number of transactions processed). A transaction is processed if either it executes completely or it is aborted. In our experiments, all transactions are assumed to be *hard* in the sense that there will be no value in completing a transaction after its deadline. Transactions that miss the deadline are aborted, and disappear from the system.

We have used transaction size (the number of data objects a transaction needs to access) as one of the key variables in the experiments. It varies from a small fraction up to a relatively large portion (10%) of the database so that conflicts would occur frequently. The high conflict rate allows synchronization protocols to play a significant role in determining system performance. We also chose the average arrival rate so that protocols are tested in a heavily loaded rather than lightly loaded system. For designing real-time systems, one must consider high load situations. Even though they may not arise frequently, one would like to have a system that misses as few deadlines as possible when such peaks occur. In other words, when a crisis occurs and the database system is under pressure is precisely when making a few extra deadlines could be most important [Abb88].

We normalize the transaction throughput in terms of data objects accessed per second for successful transactions, not in transactions per second, in order to account for the fact that bigger transactions need more database processing. The normalization rate is obtained by multiplying the transaction completion rate (transactions/second) by the transaction size (data objects accessed/transaction).

In Figure 2, the throughput of the priority ceiling protocol (C), the two-phase locking protocol with priority mode (P), and the two-phase locking protocol without priority mode (L), is shown for transactions of of different sizes. Since we chose the average arrival rate to make the system heavily loaded, both CPU and I/O were very heavily loaded when the average transaction size reaches 20. As the transaction size increases, there is little impact on the throughput of the priority ceiling protocol over a range of transaction sizes shown in Figure 2. This is because in the priority ceiling protocol, the conflict rate is determined by ceiling blocking rather than direct blocking, and the frequency of ceiling blocking is not sensitive to the transaction size.

However, the performance of the two-phase locking protocol with or without priority degrades very rapidly. This phenomenon is more pronounced as the transaction workload becomes more I/O bound, since there are few conflicts for the small transactions in the two-phase locking protocol, and the concurrency is fully achieved with an assumption of parallel I/O processing. Poor performance of the two-phase locking protocol for bigger transactions is due to the high conflict rate.

Another important performance statistic is the percentage of deadline-missing transactions, since the synchronization protocol in real-time database systems must satisfy the timing constraints of individual transaction. In our experiments, each transaction's deadline is set in proportion to its size and system workload (number of transactions), and the transaction with the earliest deadline is assigned the highest priority. As shown in Figure 3, the percentage of deadline-missing transactions increases sharply for the two-phase locking protocol as the transaction size increases. A snarp rise was expected, since the probability of deadlocks would go up with the fourth power of the transaction size [Gray81]. However, the percentage of deadline-missing transactions increases more slowly as the transaction size increases in the priority ceiling protocol. Since there is no deadlock in priority ceiling protocol, the response time is proportional to the transaction size and the priority ranking.

4. Priority Ceiling in Distributed Environments

In this section, we discuss the use of the priority ceiling approach as a basis for real-time locking protocol in a distributed environment. The priority ceiling protocol might be implemented in a distributed environment by using the global ceiling manager at a specific site. In this approach, all decisions for ceiling blocking is performed by the global ceiling manager. Therefore all the information for ceiling protocol is stored at the site of the global ceiling manager.

The advantage of this approach is that the temporal consistency of the database is guaranteed, since every data object maintains most up-to-date value. While this approach ensures consistency, holding locks across the network is not very attractive. Owing to communication delay, locking across the network will only enforce the processing of a transaction using local data objects to be delayed until the access requests to the remote data objects are granted. This delay for synchronization, combined with the low degree of concurrency due to the strong restrictions of the priority ceiling protocol, is counterproductive in real-time database systems.

An alternative to the global ceiling manager approach is to have replicated copies of data objects. An up-to-date local copy is used as the primary copy, and remote copies are used as the secondary read-only copies. In this approach, we assume a single writer and multiple readers model for distributed data objects. This is a simple model that effectively models applications such as distributed tracking in which each radar station maintains its view and makes it available to other sites in the network. For this approach to work, the following restrictions are necessary:

(1) Every data object is fully replicated at each site.

(2) Data objects to be updated must be a primary copy at the same site with the updating transaction.

(3) Every transaction must be committed before updating remote secondary copies.

Under these restrictions, the local ceiling manager at each site can enforce the priority ceiling protocol for the synchronization of not only the local data objects (primary or replicated copies), but also remote primary copies and local replicated copies. The first restriction is necessary because in a distributed database environment, holding locks across the network will occur if all the data objects requested by a transaction do not reside at the local site. If we allow each transaction to update its local copy without synchronizing with other transactions, transaction roll back and subsequent abort may result as in opumistic

concurrency control. This situation is not acceptable in real-time applications. The second restriction prevents it by providing only a single primary copy.

If we insist that copies of a data objects must be identical with respect to all references, a transaction updating the primary copy cannot commit until all the remote copies are also updated. However, this solution requires locking data objects across the network, which can lead to long durations of blocking. The third restriction solves this problem by allowing remote copies to be historical copies of the primary copy; the primary and remote copies can be updated asynchronously. The third restriction, however, may cause a temporal inconsistency, owing to the delays in the network. That is, some of the views can be out of date. Even with this potential problem of reading out of date values, the third restriction is very critical in improving the system responsiveness in distributed environments. This also solves the problem of distributed deadlock. Since we do not have deadlocks at each site, and locks are not allowed to be held across the network, we cannot have distributed deadlocks.

We have investigated the performance characteristics of the global ceiling approach and the local ceiling approach with replication in a distributed environment. The real-time database system we have prototyped for the experiment consists of three sites with fully interconnected communication network. To focus on the impact of the transaction mix and the communication cost on the number of deadline-missing transactions, we did not include any I/O cost for the experiments. In other words, a memory-resident database system in a distributed environment was simulated. As in the single-site experiments, transactions enter the system with the exponentially distributed interarrival times and they are ready to execute when they appear in the system. Update transactions are assigned to a site based on their write-set, and read-only transactions are distributed randomly. The objects updated by a transaction are chosen uniformly from the database.

Figure 4 shows the ratio between the throughput of the global ceiling approach and that of the local ceiling approach, based on different transaction mix and communication delays. Even without considering the communication delay (i.e., communication delay = 0), the local ceiling approach achieves the throughput between 1.5 and 3 times higher than that of the global ceiling approach, over the wide range of transaction mix. The reason for this difference is that the degree of concurrency among the transactions at each site can be greatly improved due to the decoupling effect of data replication. If we consider communication delays, this performance ratio will increase according to the communication delay as shown in Figure 4.

Figure 5 illustrates the ratio of the percentage of deadline missing transactions between the global and the local ceiling approach, based on different communication delays for a specific transaction mix (50% read-only and 50% update transactions). There is a significant difference between the two approaches in the number of deadline missing transactions, although the increase rate of this performance ratio varies with the communication delay. In the range of small communication delays (up to 2 time units), this ratio increases rapidly, and then rather slowly after that. As the communication delay increases, the performance ratio increases beyond 16. This implies that the global ceiling approach is more than 16 times likely to miss the real-time constraints than the local ceiling approach, for a given set of real-time transactions. Performance improvement of the local ceiling approach is more substantial with small communication delays than with large delays. This is because as communication delay increases, the concurrency achieved by the local ceiling approach is limited by the the communication cost due to data replication. Figure 6 shows the percentage of deadline-missing transactions for two specific communication delays. As shown in Figure 5, the performance difference in terms of deadline-missing transactions between two approaches increases as the communication delay increases over a wide range of transaction mix. As the proportion of read-only transactions increases, the number of deadline-missing transactions decreases since the conflict rate will decrease.

Our performance results have illustrated the superiority of the local ceiling approach over the global ceiling approach, at least under one representative distributed real-time database and transaction model. Hence, from this experimentation, we believe that, even with the potential problem of temporal inconsistency (i.e., reading out of date values), the local ceiling approach is a very powerful technique for real-time concurrency control in distributed database systems.

There are applications where a temporally consistent view is more important than just the latest information that can be obtained at each site. For example, in an application like tracking, a local track would be updated periodically in conjunction with repetitive scanning. In order to provide a temporally consistent view in a distributed environment, we can utilize the periodicity of the update transaction as a timestamp mechanism. If the system provide multiple versions of data objects, ensuring a temporally consistent view becomes a real-time scheduling problem in which the time lags in the distributed versions need to be controlled. Once the time lags can be controlled by the timestamps of data objects, transactions can read the proper versions of distributed data objects, and ensure that decisions are based on temporally consistent data.

5. Conclusions

Prototyping large software systems is not a new approach. However, methodologies for developing a prototyping environment for distributed database systems have not been investigated in depth in spite of its potential benefits. In this paper, we have presented a prototyping environment that has been developed based on the StarLite concurrent programming kernel and message-based approach with modular building blocks. Although the complexity of a distributed database system makes prototyping difficult, current implementation of the prototyping environment has proven satisfactory for experimentation of design choices, different database techniques and protocols, and even an integrated evaluation of database systems. It supports a very flexible user interface to allow a wide range of system configurations and workload characteristics. Since our prototyping environment is designed to provide a spectrum of database functions and operating system modules, it facilitates the development of multiple system instances with different characteristics without much overhead. Expressive power and performance evaluation capability of our prototyping environment has been demonstrated by prototyping a distributed real-time database system and investigating its performance characteristics.

In real-time database systems, transactions must be scheduled to meet their timing constraints. In addition, the

system should support a predictable behavior such that the possibility of missing deadlines of critical tasks could be informed ahead of time, before their deadlines expire. Priority ceiling protocol is one approach to achieve a high degree of schedulability and system predictability. In this paper, we have investigated this approach and compared its performance with other techniques and design choices. It is shown that this technique might be appropriate for real-time transaction scheduling since it is very stable over the wide range of transaction sizes, and compared with two-phase locking protocols, it reduces the number of deadline-missing transactions.

There are other technical issues associated with priority-based scheduling protocols that need further investigation. For example, the analytic study of the priority ceiling protocol provides an interesting observation that the use of read and write semantics of a lock may lead to worse performance in terms of schedulability than the use of exclusive semantics of a lock. This means that the *read* semantics of a lock cannot be used to allow several readers to hold the lock on the data object, and the ownership of locks must be mutually exclusive. Is it necessarily true? We are investigating this and other related issues using the prototyping environment.

Transaction scheduling options for real-time database systems also need further investigation. In priority ceiling protocol and many other database scheduling algorithms, preemption is usually not allowed. To reduce the number of deadline-missing transactions, however, preemption may need to be considered. The preemption decision in a real-time database system must be made very carefully, and as pointed out in [Stan88], it should not necessarily based only on relative deadlines. Since preemption *implies not only that the work* done by the preempted transaction must be undone, but also that later on, if restarted, must redo the work. The resultant delay and the wasted execution may cause one or both of these transactions, as well as other transaction to miss the deadlines. Several approaches to designing scheduling algorithms for real-time transactions have been proposed [Liu87, Stan88, Abb88], but their performance in distributed environments is not studied. The prototyping environment described in this paper is an appropriate research vehicle for investigating such new techniques and scheduling algorithms for distributed real-time database systems.

References

[Abb88] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Study," *VLDB Conference*, Sept. 1988, 1-12.

[Abb89] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *VLDB Conference*, August 1989.

[Esw76] Eswaran, K. P. et al, "The Notion of Consistency and Predicate Locks in a Database System," *Comm. of the ACM*, Nov. 1976, 624-633.

[Gray81] Gray, J. et al., "A Straw Man Analysis of Probability of Waiting and Deadlock," *IBM Research Report*, RJ 3066, 1981.

[IEEE89] *Sixth IEEE Workshop on Real-Time Operating Systems and Software*, Pittsburgh, Pennsylvania, May 1989.

[Liu87] Liu, J. W. S., K. J. Lin, and S. Natarajan, "Scheduling Real-Time, Periodic Jobs Using Imprecise Results," *Real-Time Systems Symposium*, Dec. 1987, 252-260.

[ONR89] *ONR Annual Workshop on Foundations of Real-Time Computing*, White Oak, Maryland, Oct. 1989.

[Raj89] Rajkumar, R., "Task Synchronization in Real-Time Systems," *Ph.D. Dissertation*, Carnegie-Mellon University, August 1989.

[Sha87] Sha, L., R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocol: An Approach to Real-Time Synchronization," *Technical Report*, Computer Science Dept., Carnegie-Mellon University, 1987.

[Sha88] Sha, L., R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988, 82-98.

[Shin87] Shin, K. G., "Introduction to the Special Issue on Real-Time Systems," *IEEE Trans. on Computers*, Aug. 1987, 901-902.

[Son88] Son, S. H., "Semantic Information and Consistency in Distributed Real-Time Systems," *Information and Software Technology*, Vol. 30, September 1988, pp 443-449.

[Son88b] Son, S. H., "Real-Time Database Systems: Issues and Approaches," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988.

[Son88c] Son, S. H., "A Message-Based Approach to Distributed Database Prototyping," *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, May 1988, 71-74.

[Son89] Son, S. H. and A. Agrawala, "Distributed Checkpointing for Globally Consistent States of Databases," *IEEE Transactions on Software Engineering*, Vol. 15, No. 10, October 1989, 1157-1167.

[Son90] Son, S. H., "An Environment for Prototyping Real-Time Distributed Databases," *International Conference on Systems Integration*, Morristown, New Jersey, April 1990.

[Stan88] Stankovic, J. and W. Zhao, "On Real-Time Transactions," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988, 4-18.

[Tok89] Tokuda, H. and C. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM Operating Systems Review*, Vol. 23, No. 3, July 1989.

[Wat88] Watson, P., "An Overview of Architectural Directions for Real-Time Distributed Systems," *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, May 1988, 59-65.

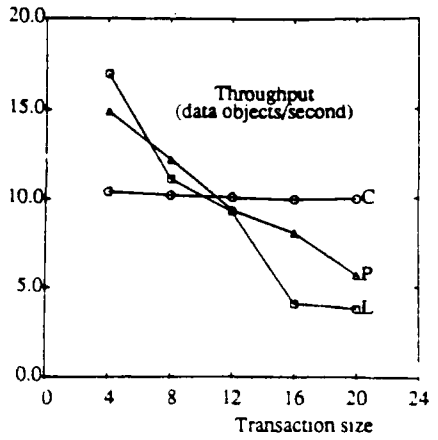


Fig. 2 Transaction Throughput.

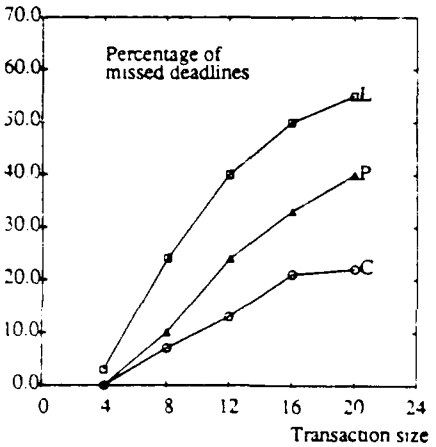


Fig. 3 Percentage of Deadline Missing Transactions.

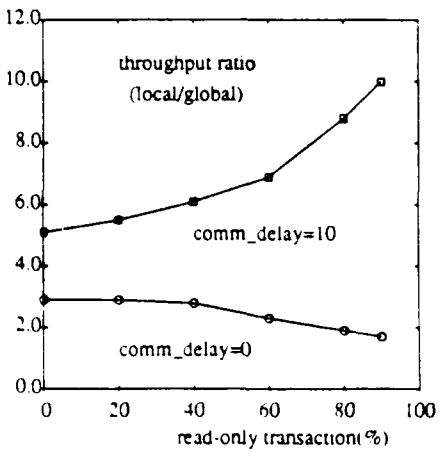


Fig. 4 Transaction Throughput Ratio

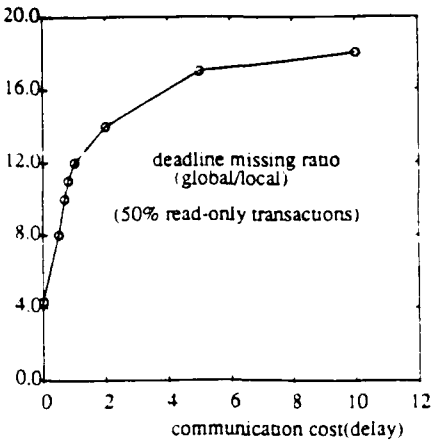
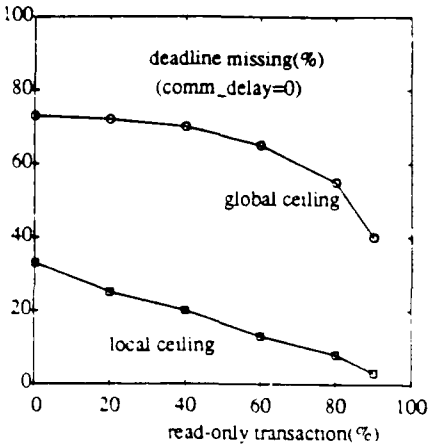


Fig. 5 Deadline Missing Ratio



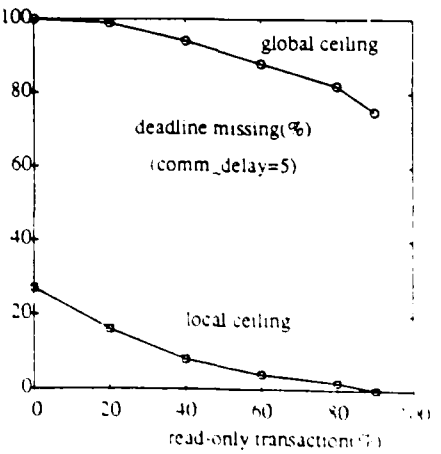


Fig. 6 Deadline Missing Transaction Percentage

DISTRIBUTION LIST

1 - 6	Director Naval Research Laboratory Washington, DC 20375 Attention: Code 2627
7 - 18	Defense Technical Information Center, S47031 Building 5, Cameron Station Alexandria, VA 22314
19	Dr. James G. Smith, Program Manager Division of Applied Math and Computer Science Code 1211 Office of Naval Research 800 N. Quincy Street Arlington, VA 22217-5000
20	Dr. Gary Koob, Program Manager Computer Science Division Code 1133 Office of Naval Research 800 N. Quincy Street Arlington, VA 22217-5000
21 - 22	R. P. Cook, CS
23	S. H. Son, CS
24	A. K. Jones, CS
25 - 26	E. H. Pancake, Clark Hall
27	SEAS Preaward Administration Files
28	Mr. Michael McCracken Administrative Contracting Officer Office of Naval Research Resident Representative 818 Connecticut Avenue Eighth Floor Washington, DC 20006

JO#3408:jame